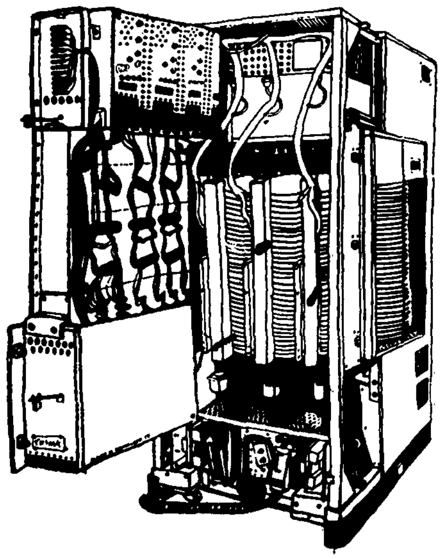
Carnegie Institute of Technology

Pittsburgh 13, Pennsylvania

216550

62 1-4
XEROX

Computation Center



Progress Report

RESEARCH AND DEVELOPMENT IN PROGRAMMING STUDY ASSIGNMENTS

Contract No. DA-36-039 SC-75081
File No. 0195-PH-58-91 (4461)
DA Task Number 3B28-04-001-01-04

I. Final Report
II. Flow Charts
III. Programming Manual

22 February 1961
for
UNITED STATES ARMY SIGNAL R&D LABORATORY
FORT MONMOUTH, NEW JERSEY

by

A. J. Perlis

Carnegie Institute of Technology
Schenley Park, Pittsburgh 13, Pennsylvania

Errata and Addenda
to the
Final Report

Contract No. DA-36-039 SC-75081
DA Task Number 3B28-04-001-01-04

October 19, 1961

by

A. J. Perlis

for

UNITED STATES ARMY SIGNAL R&D LABORATORY
FORT MONMOUTH, NEW JERSEY

Carnegie Institute of Technology
Schenley Park, Pittsburgh 13, Pennsylvania

Errata and Addenda to the Final Report DA-36-039 SC-75081

art I.

1. p2, L 7 Amplification

The input is in "string" format, i.e., as a concatenation of cha

from the 20^L alphabet. The only formal structure to such a string

it has a beginning (1st character) and an end(last character).

The transformation to "tree" form is accomplished by differentia

behavior of the 20^ translator on observing certain character combi

in the input string. The "tree" form has punctuation introduced --

and")" which permits the decomposition of the input string into lev

(and thereby sub-levels, etc.)

2. p2, L 12

A terminal such. $\rightsquigarrow$[1] A terminal such time.

3. p3, L 16 Amplification

"Atomic" machine instructions are those wired in commands, or sc

even those precomposed in fixed order from such commands.

4. p5, L 6 Amplification

::= is a metalinguistic connective meaning " is defined as", i.e.

a::=b means a is defined as b.

5. p8, L 14

<identifier>::=<letter> ... $\rightsquigarrow$ <letter>|<identifier><letter>|
<identifier><digit>|
<special register identifier>

6. p 11, L 4-[2]

(7 $\rightsquigarrow$ (3

---

1. a $\rightsquigarrow$ b means a is replaced by b
2. L 8- means 8th line from the bottom

7. p 12, L 9

(8 $\rightsquigarrow$ (7 Ref 7 on pg 5 herein

8. L 5- Amplification

Element sequencing is explained in detail in (7. Suffice it to say here that its purpose is to examine symbols <u>independantly</u> of the structures in which they are imbedded. Word and list sequencing on the other hand are structure dependant.

9. p17, L 12 insrt (k,v) $\rightsquigarrow$ isrt (k,v)

Amplification: if k is missing, it is assumed to be 1.

10. p18, L1 $\rightsquigarrow$ 3.3 Arithmetic Expressions

11. L 4 ⟨primary⟩ ::= .... $\rightsquigarrow$

⟨primary⟩ ::= ... |(⟨arithmetic expression⟩)|
⟨row primary⟩|⟨column primary⟩

note: for use of ⟨row primary⟩, and ⟨column primary⟩ see section 5.2.3.4 pg 3 herein

12. p28, L 13 Amplification

32, 767 = $2^{15}-1$ and obviously assumes a computer with that size memory.

13. L 22 ff Amplification

3.7.2.1 The vertical bar here used is not to be confused with the metalinguistic connective "or". This meaning is given in 3.7.3.
In expression, to right of |, parentheses mean "contents of".

14. p29, L 4 about them. $\rightsquigarrow$ about them (see page 43).

15. p37, L 3 Statement if $\rightsquigarrow$ statement it

16. p41, L 7 These examples correspond $\rightsquigarrow$

These examples of procedure statements correspond to examples of procedure declarations given in section 5.4.2.

17. p42, L 6

Procedure heading $\rightsquigarrow$ procedure declaration heading

18. p44, L 1 4.9.9. → 4.9.1.

19. L 11 (A C t, P ... $\rightsquigarrow$ (A [t, P ...

20. p47, L 2 insertion after

⟨storage function⟩::=⟨arithmetic expression⟩

21. L 6 insertion after

⟨bound pair set⟩::= [⟨bound pair list⟩] ⟨storage function⟩

| [⟨bound pair list⟩]

22. L 7 [⟨bound pair list⟩] $\rightsquigarrow$ [⟨bound pair set⟩]|

23. L 15 insertion after

real array Z [1:n, 1:n] $(n-\xi+1) \times (\xi-1)+\eta-1$

see 5.2.3.4. for the semantics of ⟨storage function⟩.

24. p48, L 2 insertion after

5.2.3.4. Storage function:

An arithmetic expresssion which is a mapping of a two dimensional array of information into a one dimensional ordered set of information. A systemactic method of specifying the row and column indices of the elements in a two dimensional array has to be made. While this can be done by defining the storage function as a procedure with formal parameters this complexity is absurd since very few uses of storage functions will require the procedure mechanism. Thus one should specialize on two characters representing the row, e.g., ξ, and column, e.g., η, indices for which substitution is made by evaluation of the expression each time and element of the array is to be identified. The definition of these special characters can be imbedded in the syntax by adding to the definition of arithmetic expressions the syntactic types ⟨row primary⟩ and ⟨column primary⟩ as in section 3.3.1. The characters ξ and η (or their equivalents) can be added to the 20 ^L alphabet.

25. p50, L 14 ; ⟨specification part⟩ ... $\rightarrow$
⟨value part⟩ ⟨specification part⟩

26. p 52, L 14 The use of ... $\rightarrow$
The use of Procedure Statements (see 4.7) and / or Function Designators (see 3.2) ...

27. p 55, L 9, L 11 delete |

Part III.

28. p 99, L 2- discussed and $\rightarrow$ discussed (see Part I) and

29. p 100, L 1 delete

30. L 1- or combination $\rightarrow$ or combination of :

31. p102, L 5 101.1 $\rightarrow$ 101

32. p112, L 5 ... ≤ x < //) (divides ... $\rightarrow$
... ≤ x < //) ∧ ¬ (divides ...

33. p112, L 7 <u>begin</u> if (divides ... $\rightarrow$
<u>begin</u> <u>if</u> ¬ (divides ...

34. p113, L 1 delete

35. L 5 insert

where * is used to represent multiplication

36. p118, L 10 insert

Here, the last two digits repres the exponent, p, of a floating point number where the exponent is represented as p + 50.

37. p 120, L 8 ...∧b > ia ia > ... $\rightsquigarrow$

...∧b > ia ∧ ia > ∧a

38. L 10 ia ba $\rightsquigarrow$ ia ∨ ba

39. L 11 ia) ¬a → ia) ∧ ¬a

40. L 12 := (ba bb); $\rightsquigarrow$

:= ¬(ba ∧ bb);

41. p 122, L 22 insert

The notation ...... represents intervening statements immaterial to the points under discussion.

42. p 123, L 12 := C 2 - B $\rightsquigarrow$:=C + 2 - B;

43. p 125, L 7 insert after

where $T_1(X) = 3\ p^2 + p$

and $T_2(X) = p^3 - 2\ p$

44. L 14 R:=p ... $\rightsquigarrow$ R:= p↑2 + 3 x p;

45. L 14 R:=p ... $\rightsquigarrow$ S:= p↑3 - 2 x p;

46. p 130, L 3- VA [i] 2 VB [i] $\rightsquigarrow$ VA [i] + VB [i];

47. p 131, L 14 amplification

p = 101 (2) and p = 5 (10)

mean

p = 101 in binary and p= 5 in decimal, respectively.

48. Bibliography append

7. Evans, A. Jr.; Perlis, A. J.; Van Zoeren, H.; The use of threaded lists in constructing a combined ALGOL and machine-like assembly processor. Comm. ACM 4 (January 1961, 36-39.

Part II.

49. p 58, L 1 a 20 $^\wedge$ L a 20 $^\wedge$ L running

50. p 59, bottom append

where ␣ denotes a blank, and Bt denotes the block tag.

51. p 60 L 8 insert

where CBT is the Code Block Table.

52. L 6- notation. In

$\rightsquigarrow$ notation and θ is some (at most) binary operator.

53. L 5- missing.

$\rightsquigarrow$ missing depending on the operator.

54. p61 L 1, 2 delete

55. L 5 or all $\rightsquigarrow$ of all

56. L 8 insert

− subtraction

57. L 20 insert

J jump

go to transfer

58. p 62, L 9- the right of $\rightsquigarrow$ the left of

59. p 64 page number should be 63.

60. p 63, L 5 the θ code $\rightsquigarrow$ the θS code

, L 7 " "

, L 14 in IS $\rightsquigarrow$ in the input sequence IS

, L 18 in, I C and OC $\rightsquigarrow$ in , IS and OS.

, L 21 will be $\rightsquigarrow$ will often be

, L 2- of the form $\rightsquigarrow$ of that form

61. p 64, L 8- insert

Note: In the following $\mathcal{O}$ and $\mathcal{OS}$, I and IS, refer interchangeably to output and input sequences, respectively. Similarly isrt and insrt are used interchangeably.

62. bottom

Note: '<u>Blend</u>' is inserted by the main declaration and means block end.

63. p 65, L 6 insert

<u>MT</u> $\mathcal{F}$ is a transfer to the routine $\mathcal{F}$ such that $\mathcal{F}$ upon completion returns to the statement following the statement <u>MT</u> $\mathcal{F}$.

,L 1- replace by

CBT $[,*]$:= Bt, $|$ (γ), (S)

isrt (o$[,\phi]$), next (o$[,\phi]$) := '<u>blend</u>'

Notes: The vertical $|$ is used in the sense of address expressions (see section 3.7).

The routines isrt(α) and next (β) are defined in reference 3 and essentially cause an empty site to be inserted following the site α; and the site one beyond the pointer for β to be referenced, respectively.

64. p 66, L 2-

Note: the digit 1 is occasionally used in place of the value <u>true</u>; and similarly so for 0 and <u>false</u>.

65. p 6$\overset{7}{\cancel{8}}$, L 5 next (O$[,\phi]$: = $\rightsquigarrow$ next (O$[,\phi]$):=

66. p 68, L 3 K:=1 $\rightsquigarrow$ K:=K+1

$[K] \rightsquigarrow [K]$

67. p 68, L 5 Note:

σ stands for the operator which is to be compiled in this line of code. It is generated in the expression analyzer.

68. p 68, L 13 <u>until</u> 12 $\rightarrow$ <u>until</u> 13

69. p 68, L 15,ff. should read

<u>Comment</u>. 'declaration' handles lists of identifier possessing the same declared attributes and, in particular, handles array declarations. In case of

70. p 68, L 5- * column $\rightarrow$ * column, for a rectangular array.

71. p 68, L 5- insert:

(2) of 1 dimension, A [m:n] there is computed

space := abs (n-m+1)

base := storage base - 1

storage base := storage base + space

and base is stored in the address assigned to A. The mapping function for A[i] is then base + i.

72. p 70, L 14 true $\rightarrow$ <u>true</u>

~~73. p 70, L 10- Note:~~

74. p 71, L 3 MT ¢ $\rightarrow$ MT [¢]

75. p 71, L 9, 10

String transfer is the table of Macro identifiers $\rightarrow$ String transfer ~~noun~~ norm is the number of characters in the Macro.

76. p 71, L 11 insert after <u>end</u>:

(this indicates the end of the Macro declaration)

77. p 71, L 12 delta [13] $\rightarrow$ delta [12]

78. p 71, L 4- should read

J := line number (library table (A))

79. p 71, L 3- should read

J := field (2, library table [J])

80. p 72, L 1 should read

next (next (I [,ȼ])) := <u>procedure</u>

81. p 72, L 3 Note:

copy serves to copy the list whose starting address is in J into I [,ȼ]

82. p 72, L 7- insert after 'there':

is in the output sequence

83. p 72, L 3- a 0 (1) $\rightsquigarrow$ a = 0 (1), respectively

84. p 72, L 1- b 0 (1) $\rightsquigarrow$ b = 0 (1), respectively

85. p 73 replaced entirely by: (See page 10.)

The catalogue of actions in the four cases are given in the following table:

for A i, j

	Declaration	Call
	a = 0	a = 0, b = 0
Pass 1	A : = base $\overline{A + 1}$: = Column	generate code for A + j + $\overline{A+1}$ * i
Pass 2	no action	execution of above
	a = 0, b = 1	
Pass 1	A : = base A + 1: = column	generate code for: A + j + A+1 * $\overline{i}$ + β_s if local identifier or compute β_s in B.A. routine and store β_s in I. and compute A + j + A+1 * i + I.
Pass 2	no action	execution of above
	a = 1, b = 0	(dynamic declaration, not inside procedure) code for A + j + $\overline{A+1}$ * i
Pass 1-generate code for A: = base	A+1: = column	
Pass 2	execution of above	execution of above
	a = 1, b = 1	
Pass 1-generate code for A[β_s]: = base	A + 1 [β_s]: = column	A[β_s] + j + $\overline{A+1}$[β_s] * i if local identifier or compute β_s in B.A. routine Store β_s in I.
Pass 2	execution of above	execution of above

86. p 74, L 3 Note:

P [i] refers to the i^{th} line of a table containing either an operator or operand of the expression being analyzed. There is assumed to exist an **internal** list of operators whose order specified the hierarchy of their execution order, e.g.,

$\uparrow$, *, /, +, -

means $\uparrow$ done before * before /, etc.

87. p 74, L 12 delimiter $\rightsquigarrow$ '('

88. p 74, L 6- := digit $\rightsquigarrow$:= decimal digit

:= 10 $\rightsquigarrow$:= $_{10}$ (the notation for base 10)

89. p 75, L 5- Note:

m.j. means code line m. j is an $\overset{\rightsquigarrow}{\text{index}}$

90. p 77 L 4 e(1); $\rightsquigarrow$ e(1));

91. p 77 L 3 ff Note:

In each **instance** the argument ,α, of code line (α) should be delimited by ' ' to indicate the nature of the substitutions being employed. See pg 66

92. p 77 L 7 =':' P[i+ 2] $\rightsquigarrow$ =':' $\wedge$ P[i + 2]

93. p 77, L 12 :=, $\rightsquigarrow$:= ',''

94. p 77, L 14 should read

<u>if</u> P[i] = Expression terminal [k] <u>then</u> <u>go to</u> |(expression analyzer)

95. p 77, L 19 I. Q. $\rightsquigarrow$ I. σ.

96. p 78, L 1 I. Q $\rightsquigarrow$ I. σ.

97. p 78, L 2 (R [k]) $\rightsquigarrow$ (L [k])

98. p 78, L 3 <u>go to</u> (L [k]) $\rightsquigarrow$ <u>go to</u> |(L [k]));

99.	p 79, L 10	should read
		else go to |(expression analyzer) end
100.	p 80, L 13	MT.BA → MT. / (BA)
	, L 1-	"
	p 81, L 9-	"
	p 81, L 1-	"
101.	p 81, L 2-	EX 16 → E x 20
102.	p 81, L 3-	P [i-5] → (P[i-5])
103.	p 82, L 2	EX 16 → EX 20
104.	p 83, L 10-	should read
		go to (j$_\nu$)
105.	p 86, L 4	begin : I → begin I
106.	p 86, L 13	for list → for, list
107.	p 86, L 2 -	next (~~[,¢]~~) → next (0 [,¢])
108.	p 88, L 14	should read
		i = generated by comma
109.	p 89, L 11-	$\frac{x}{1} \to (\frac{x}{1}$
110.	p 90, L 9 -	(T [H]) → (IT [H])
111.	p 90, L 8 -	should read
		Bl 2: if marker (IT [H]) ≠ '('
112.	p 90, L 7-	should read
		then begin H: = H $^-$1; go to B12 end
113.	p 90, L 6 -	number > → number [⁄] >
114.	p 91, L 1ff	Note:
		T and IT are used interchangeably for the identifier table.

115.	p 91, L 13 –	norm field → norm (field
116.	p 92, L 2 –	is A → is in A
117.	p 94, L 1	operand → operator
118.	p 94, L 7, L 8	MΓ [z] → MI [ƶ]
119.	p 94, L 17	seq N (G, → seqw (G,
		seqw N (V, → seqv (V,
120.	p 96.1 L 4 –	(–.l.e → (–.L.e
121.	p 96.2 L 5	go to → $\underline{\text{go to}}$
122.	p 96.2, L 7, L 9	code line (I.Ω. → code line (I.σ.

INTRODUCTION

A single programming language is described. Though it has, from the programmer's point of view, three forms, there is in reality only one language and only one processor for producing machine code.

In keeping with historical precedence the three forms as mentioned are:

(i) An algebraic language

(ii) A symbolic machine-like coding language

(iii) A symbol manipulation list processing language

The algebraic language is ALGOL[1] with some trivial extensions

The symbolic machine-like language is like TASS[2]

The symbol-manipulation language is like Threaded Lists[3]

Nevertheless the three are described as one language using the same syntax description; and there is only one processor.

The name given to the language is 20^L,[4] that of the processor is 20^P.

The report that follows is divided into sections that:

Define the Language Syntax of 20^L (Part I)

Define the Process Syntax of 20^P (Part II)

Define the Use of 20^L (Part III)

Naturally, a particular computer must be used as a model for (ii) above; and some of the Process Syntax will be similarly machine dependent. The machine used as a model--where so necessary--is the Bendix G 20[5].
Effort has been made to keep such reference to a minimum.

Associated with the processor are modes of operation which, of course, are machine dependent. One such is described in Part III. The organization of 20$^\wedge$P is intended to permit flexible operating schemes, consequently the design of the processor is described in terms of actions taken at various phases of the translation process.

The phases are directly related to "passes" through the code sequences admitted to and generated by the system. Three passes are involved in the description of the translation process:

i) Reduction from string representation to tree representation *.

ii) Reduction from tree representation to machine code sequence representation.

iii) Reduction of machine code sequence representation to output data representation.

With respect to each of the syntactic units of 20$^\wedge$L there is associated a "time" of <u>declaration</u> and a "time" of <u>call</u>. With each of these there is a <u>first</u>, an <u>intermediate</u>, and a <u>terminal</u> such

20$^\wedge$P generates tables of relations between properties of syntactic units in the three representations. Operations on these tables do not constitute a pass in the sense above. Instead they are imbedded in <u>transition</u> phases between the passes.

The following diagram will be useful in describing processing tasks on the various syntactic units:

Pass	Declaration	Call	Transition
1			
2			
3			

first	intermediate	terminal

Thus within the descfiptions that follow, the notation Di3 will refer to an intermediate declaration processing during pass 3.

---

* Expressions are handled-out of deference to storage efficiency--in a 3 address block form, rather than a pure tree form.

Part I. The Language: $20^{\wedge}L$

The form of the description that follows borrows heavily on the report of ALGOL 60.(1 Indeed, since the ALGOL 60 language is so much an integral part of the $20^{\wedge}L$ (+, parts of the report are reproduced almost in toto in the sequal. No further mention will be made within those portions of their source.

A word should be said about representation. The characters of the alphabet are assumed to be distinct and individually recognizable. A unique collection of characters underlined is assumed to be a unique character of the alphabet.

What representation one may choose to use on restricted character input devices is not the concern of this report.

The purpose of the algorithmic language, $20^{\wedge}L$ is to describe computational processes. The basic concepts used for the description of calculating rules are

(i) "atomic" machine instructions

(ii) list instructions containing as constituents numbers, symbols, variables, functions, and relations.

(iii) the well-known arithmetic expression containing as constituents numbers, variables, functions and relations.

From such basic units are compounded, by applying rules of arithmetic composition, self-contained units of the language--explicit formulae--called assignment statements.

To show the flow of computational processes, certain control statements and statement clauses are added which may describe, e.g.,

(+ The definition of ALGOL 60 was--in part--the responsibility of the senior project member.

alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these control statements that one statement refer to another, statements may be provided with labels. Sequences of statements may be combined into compound statements by insertion of statement brackets.

Statements are supported by declarations which are not themselves computing instructions, but inform 20$\wedge$P of the existence and of certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers, or even the set of rules defining a function. Each declaration is attached to and valid for one compound statement. A compound statement which includes declarations is called a block.

A program is a self-contained compound statement, i.e., a compound statement which is not contained within another compound statement and which makes no use of other compound statements not contained within it. (*

In the sequel the syntax and semantics of the language will be given.

(* Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

1.1. Formalism for Syntactic Description

(6

The syntax will be described with the aid of metalinguistic formulae. Their interpretation is best explained by an example

<ab> ::= (| [| <ab> (| <ab> <d>

Sequences of characters enclosed in the brackets < > represent metalinguistic variables whose values are sequences of symbols. The marks ::= and | (the latter with the meaning of <u>or</u>) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable <ab>. It indicates that <ab> may have the value (or [or that given some legitimate value of <ab>, another may be formed by following it with the character (or by following it with some value of the variable <d>. If the values of <d> are the decimal digits, some values of <ab> are:

[(((1(37(

(12345(

(((

[86

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e., the sequences of c aracters appearing within the brackets < > as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

Definition: <empty> ::=
(i.e., the null string of symbols).

2. Basic Symbols, Identifiers, Numbers, and Strings.

Basic Concepts.

The reference language is built up from the following basic symbols:

<basic symbol> ::= <letter> | <digit> | <logical value> | <delimiter> | <continuation mark>

2.1. Letters

<letter> ::= a | b | c | d | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e., character not coinciding with any digit, logical value or delimiter).

Letters do not have individual meaning. They are used for forming identifiers and strings (cf. sections 2.4. Identifiers, 2.6. Strings).

2.2.1. Digits

<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<decimal digit> ::= <octal digit> | 8 | 9

Decimal digits are used for forming numbers, identifiers, and strings.

2.2.2. Logical Values

<logical value> ::= true | false

The logical values have a fixed obvious meaning.

2.3. Delimiters

<delimiter> ::= <operator> | <separator> | <bracket> | <declarator> |
    <specificator>

<operator> ::= <arithmetic operator> | <relational operator> |
    <logical operator> | <sequential operator> | <list operator>

<arithmetic operator> ::= + | - | x | / | ÷ | ↑

<relational operator> ::= < | ≤ | = | ≥ | > | ≠ |

<logical operator> ::= ≡ | ⊂ | ∨ | ∧ | ¬ | $

<list operator> ::= * | ¢ | ⁄ | ↓ | p

<instruction operator bound> ::= | | →

<sequential operator> ::= go to | if | then | else | for | do

<separator> ::= , | . | 10 | : | ; | := | ※ | step | until | while | comment

<bracket> ::= (|) | [|] | ‘ | ’ | begin | end

<declarator> ::= own | Boolean | logical | integer | octal | real | array |
    switch | procedure | index | macro | parameter | equivalent | library |
    constant | list

<specificator> ::= string | label | value

Delimiters have a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel. Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

<continuation mark> ::= ↓

The continuation mark is used in the case of symbolic machine code as a punched-card oriented convention specifying that the instruction punched on the card requires (at least) one more card to complete its description.

For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of basic symbols:	is equivalent with
; comment <any sequence not containing ;>	;
begin comment <any sequence not containing ;>;	begin
end <any sequence not containing end or ; or else>	end

By equivalence is here meant that any of the three symbols shown in the right-hand column may, in any occurrence outside of strings, be replaced by any sequence of symbols of the structure shown in the same line of the left-hand column without any effect on the action of the program.

2.5. Identifiers

2.4.1. Syntax

<identifier> ::= <letter>|<identifier>|<identifier>|<digit>|
<special register identifier>

2.4.2. Examples

q

Soup

V17a

a34kTMNs

MARILYN

2.4.3. Semantics

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf., however, section 3.2.4. Standard Functions).

However, the use of symbolic machine code is more easily mixed with the algebraic and list language if certain machine registers are recognized by identifiers fixed by convention. Thus, e.g.,

<special register identifier> ::= ACC | MQ | CA | LR

The above is intended as an example and is clearly machine dependent.

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7. Quantities, Kinds and Scopes, and section 5. Declarations).

2.5. Numbers

2.5.1. Syntax

<digit> ::= <decimal digit>

<unsigned integer> ::= <digit>|<unsigned integer> <digit>

<integer> ::= <unsigned integer> | + <unsigned integer> | - <unsigned integer>

<decimal fraction> ::= .<unsigned integer>

<exponent part> ::= $_{10}$<integer>

<decimal number> ::= <unsigned integer> | <decimal fraction> |<unsigned integer><decimal fraction>

<unsigned number> ::= <decimal number> | <exponent part> | <decimal number><exponent part>

<number> ::= <unsigned number> | + <unsigned number> | - <unsigned number>

<unsigned octal integer> ::= <octal digit> | <unsigned octal integer> <octal digit>

<octal integer>::= (8) <unsigned octal integer> |+(8)<unsigned octal integer> | -<unsigned octal integer>

2.5.2. Examples

0	-200.084	$-.083_{10}-02$	(8) 1234
177	$+07.43_{10}8$	$-_{10}7$	(8) 77777
.5384	$9.34_{10}+10$	$_{10}-4$	
+0.7300	$2_{10}-4$	$+_{10}+5$	

2.5.3. Semantics

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10. Octal numbers are used only with machine assembly code.

2.5.4. Types

Integers are of type <u>integer</u>. All other numbers are of type <u>real</u> (cf. section 5.1. Type Declarations).

2.6. Strings

2.6.1. Syntax

```
<proper string> ::= <any sequence of basic symbols not containing
        'or'> |<empty>
<open string> ::= <proper string> |'<open string>'|
                        <open string><open string>
<string> ::=  '<open string>'
```

2.6.2. Examples

'5k,,-'[[['∧ =/:'Tt''

'.. This ⁎ is ⁎ a ⁎'string''

2.6.3. Semantics

In order to enable the language to handle arbitrary sequences of basic symbols the string quotes 'and' are introduced. The symbol ⁎ denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. sections 3.2. , Function Designators and 4.7. Procedure Statements).

2.7. Quantities, Kinds and Scopes

The following kinds of quantities are distinguished: simple variables, arrays, lists, labels, switches, macros, and procedures.

The scope of a quantity is the set of statements in which the declaration for the identifier associated with that quantity is valid, or, for labels, the set of statements which may have the statement in which the label occurs as their successor.

2.8. Values and Types

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), an ordered set of strings (special case: a single string), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1.).

The value of a list identifier is the ordered set of values of the corresponding list. These values are ordered by the element sequencing rule for lists (3.8...)

The various "types" (integer, real, Boolean) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

2.9 Threaded Lists

Threaded lists (Tlists) have been reported on elsewhere (7. The following points are, however, basic to the material of the report. An information site is a place in the machine that can be occupied by i) a symbol, ii) a Tlist, iii) an address specifying the site of a symbol.

Initially a Tlist named NAME occupies two sites denoted by: NAME: (,). One information site, denoted by ,) is available in each empty Tlist of the form (,).

New blank sites may be added by an insert operation. Thus:

$$(,\ ) \xrightarrow[\text{insert}]{} (,\ ,\ ) \xrightarrow[\text{insert}]{} (,\ ,\ ,\ ).$$

In a blank site, an empty Tlist may be inserted. Thus:

$$(,\underset{\uparrow}{\ } ,\ ) \quad \text{list} \quad (,\underset{\uparrow}{(} ,\ ),\ ).$$

In order to access information in a Tlist each list may be sequenced at any time in one of three ways[8]: word, element, and list. Furthermore, if X is the name of a particular Tlist, X¢ specifies its site currently under scan. X* specifies the "next" site to that currently under scan, and makes it the current one. n(k,X¢) specifies the kth next site to that currently under scan but does not change the significance of X¢. h(X¢) specifies the site of the innermost "(" of the pair " (" and ") " which enclose X¢. Similarly the kth head of X¢ by $u(k,\text{X¢}) = \underbrace{h(h(h\ldots h}_{k}(\text{X¢}))\ldots)$. The operators h and u are themselves independent of the sequencing mode employed.

Word sequencing is a left-right sequencing with exactly one stop at each site. List sequencing is a left-right sequencing with stops only at sites which are on the same level. Element sequencing is a left-right sequencing with stops only at sites where symbols or indirect referents may occur. An example will clarify:

```
                  Tlist:  (,(, ), ,(,(,(, , )), , ), )
word sequence:             1 2  3 4 5 6 7 8  9 10 11
list sequence:             1    2 3              4
element sequence:            1  2      3 4  5 6  7
```

Two lists may be combined in several ways to form new lists. Copy (x,y) copies the list x into the list site y. Thus:

$$\underset{X}{(\ldots)} \quad \text{and} \quad \underset{y}{(,(,\ ),\ ,\ )}$$

gives for copy (X,y): $(,(,(\ldots)),\underset{\uparrow}{,}\ ,\ )$. Join (z,x,y) forms z: (x,y). Appndr (x,y) forms, using the above example, (,(,), , ,(...)).

The $\longrightarrow$ operation substitutes symbols and control characters into sites from other sites. It is a special copy working on the micro scale of a site.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational, expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential, operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

<expression> ::= <arithmetic expression> <Boolean expression> <logical expression> <designational expression> <address expression>

3.1. Variables

3.1.1. Syntax

<variable identifier> ::= <identifier>

<simple variable> ::= <variable identifier>

<subscript expression> ::= <arithmetic expression>

<subscript list> ::= <subscript expression> | <subscript list>, <subscript expression>

<array identifier> ::= <identifier>

```
<subscripted variable> ::= <array identifier> [<subscript list>]
<list variable> ::=  <list identifier> [<list subscript list>]
<list identifier> ::=  <identifier>
<list subscript list> ::=  <fore list subscript expression>, <aft list
                                                     subscript expression>
<variable> ::=  <simple variable> | <subscripted variable> | <list variable>
```

3.1.2. Examples

```
epsilon                    R[,¢]
detA                       X[p √, *]
a17
Q[7,2]
x[sin(n x pi/2), Q[3, n, 4]]
```

3.1.3. Semantics

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2.). The type of the value of a particular variable is defined in the declaration for the variable itself (cf. section 5.1. Type Declarations) or for the corresponding array identifier (cf. section 5.2. Array Declarations).

3.1.4. Subscripts

3.1.4.1. Subscripted variables designate values which are components of multidimensional arrays (cf. section 5.2. Array Declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets [ ]. The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. section 3.3. Arithmetic Expressions).

3.1.4.2. Each subscript position acts like a variable of type integer and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2. Array Declarations).

3.1.5. List Indices

3.1.5.1. List variables designate components of lists. Each list expression of the subscript list occupies one subscript position of the subscripted list variable, and is called a list subscript. The complete list of subscripts is enclosed in the subscript brackets [ ]. The list component referred to by a subscripted variable is specified by the action of the list sequencing mode currently operative over the list named. The value of the subscript is defined only if the sequencing action does not exhaust the list elements. Should exhaustion occur before the list component is encountered, control transfer within the program will occur.

3.2. Function Designators

3.2.1. Syntax

```
<procedure identifier> ::= <identifier>
<actual parameter> ::= <string> |<expression> |<array identifier>|
    <switch identifier> |<procedure identifier>
<letter string> ::= <letter> |<letter string> <letter>
<parameter delimiter> ::= , |) <letter string> : (
<actual parameter list> ::= <actual parameter>|
    <actual parameter list> <parameter delimiter>
    <actual parameter>
```

<actual paremeter part> ::= <empty> | (<actual parameter list>)

<function designator> ::= <procedure identifier>
<actual parameter part>

3.2.2. Examples

sin(a-b)

J(v+s,n)

R

S(s-5) Temperature:(T) Pressure:(P)

Compile(‘:=’)Stack: (Q)

3.2.3. Semantics

Function designators define sequencing rules, single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration(cf. section 5.4. Procedure Declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7. Procedure Statements. Not every procedure declaration defines the value of a function designator.

3.2.4. Standard functions

3.2.4.1. Standard Arithmetic Functions

Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures. It is recommended that this reserved list should contain:

abs(E)	for the modulus (absolute value) of the value of the expression E
sign(E)	for the sign of the value of E(+1 for E>0,0 for E=0, -1 for E<0)
sqrt(E)	for the square root of the value of E
sin(E)	for the sine of the value of E
cos(E)	for the cosine of the value of E
arctan(E)	for the principal value of the arctangent of the value of E

ln(E)	for the natural logarithm of the value of E
exp(E)	for the exponential function of the value of E (e^E).

These functions are all understood to operate indifferently on arguments both of type real and integer. They will all yield values of type real, except for sign(E) which will have values of type integer. In a particular representation these functions may be available without explicit declarations (cf. section 5. Declarations).

3.2.4.2. Standard List Procedures

next(v)	for the extraction of the next list component from v without advancing the sequence marker
list(v)	for the insertion of an empty list into the site v
insrt(k,v)	for the insertion of k empty sites immediately following v
def(w,v)	for the dynamic definition of a list w as v
copy(v,w)	for the creation of a list w which is, except for linking addresses, identical to the list v.
seq(e,v,w)	for the sequencing through list v in mode e with exit to w on completion

These functions operate on lists according to the formation and sequencing rules regarding lists (cf. section 3.8)

3.2.5. Transfer Functions

It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely

entier(E)

which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of E

3.3. Arithmetic Expressions

3.3.1. Syntax

```
<adding operator> ::=  + | -

<multiplying operator> ::=  x | / | ÷

<primary> ::=  <unsigned number> | <variable> |
     <function designator> | (<arithmetic expression>)

<factor> ::=  <primary> | <factor> ↑ <primary>

<term> ::= <factor> | <term> <multiplying operator> <factor>

<simple arithmetic expression> ::= <term> |
     <adding operator> <term>  | <simple arithmetic expression>
     <adding operator> <term>

<if  clause> ::=  if  <Boolean expression>then

<arithmetic expression> ::=  <simple arithmetic expression> |
     <if clause> <simple arithmetic expression>else
     <arithmetic expression>
```

3.3.2. Examples

Primaries

$7.394_{10}{-8}$

sum

w[i+2,8]

cos(y+zx3)

(a-3/y+vu ↑8)

Z[↙, ¢]

Factors:

omega

sum ↑cos(y+z x 3)

$7.394_{10}{-8}$ ↑w[i+2,8] ↑(a-3/y+vu ↑8)

Terms:

U

omega x sum ↑cos(y+z x3)/7.394$_{10}$-8 ↑w[i+2,8] ↑

(a-3/y+vu-8)

Simple arithmetic expression:

U-Yu+omega x sum ↑cos (y+z x 3)/7.394$_{10}$-8 ↑w[i+2,8] ↑

(a-3/y+vu ↑8)

Arithmetic expressions:

w x u-Q(S+Cu)↑2

if q>0 then S+3 x Q/A else 2 x S+3 x q

if a<0 then U+V else if a x b>17 then U/V else if

k≠y then V/U else 0

a x sin(omega x t)

0.57$_{10}$12 x a[N x (N-1)/2, 0]

(A x arctan(y) + Z)↑(7+Q)

if q then n-1 else n

if a<0 then A/B else if b=0 then B/A else(z x R[↙, **])

3.3.3. Semantics

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (cf. section 5.4. Procedure Declarations) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4. Boolean Expressions). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence from left to right until one having the value true is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood). The construction:

else <simple arithmetic expression>

is equivalent to the construction:

else if true then <simple arithmetic expression>

3.3.4. Operators and Types

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types real or integer (cf. section 5.1. Type Declarations).

List variables occurring in simple arithmetic expressions must of course, refer to that part of their information content which is of type real or integer. The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1. The operators +, -, and x have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be integer if both of the operands are of integer type, otherwise real.

3.3.4.2. The operations <term>/<factor> and <term> ÷ <factor> both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus for example

$$a/b \times 7/(p-q) \times v/s$$

means

$$((((a \times (b^{-1})) \times 7) \times ((p-q)^{-1})) \times v) \times (s^{-1})$$

The operator / is defined for all four combinations of types <u>real</u> and and <u>integer</u> and will yield results of <u>real</u> type in any case. The operator $\div$ is defined only for two operands both of type <u>integer</u> and will yield a result of type <u>integer</u> defined as follows:

$$a \div b = \text{sign}(a/b) \times \text{entier}(\text{abs}(a/b))$$

(cf. sections 3.2.4 and 3.2.5).

3.3.4.3. The operation <factor>↑<primary> denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example,

$2\uparrow n\uparrow k$ means $(2^n)^k$

while

$2\uparrow(n\uparrow m)$ means $2^{(n^m)}$

Writing i for a number of <u>integer</u> type, r for a number of <u>real</u> type, and a for a number of either <u>integer</u> or <u>real</u> type, the result is given by the following rules:

$a\uparrow i$ If $i>0$, $a \times a \times \dots \times a$ (i times), of the same type as a.

If $i=0$, if $a\neq 0$, 1, of the same type as a.

if $a=0$, undefined.

If $i<0$, if $a\neq 0$, $1/(a \times a \times \dots \times a)$ (the denominator has i factors), of type <u>real</u>.

if $a=0$, undefined.

a↑r If a>0, exp(r x ln(a)), of type __real__.

If a=0, if r>0, 0. 0, of type __real__.

if r≤0, undefined.

If a<0, always undefined.

3.3.5. Precedence of operators

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.3.5.1. According to the syntax given in section 3.3.1 the following rules of precedence hold:

first	↑
second:	x/÷
third:	+ -

3.3.5.2. The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6. Arithmetics of __real__ quantities

Numbers and variables of type __real__ must be interpreted in the sense of numerical analysis, i.e., as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4. Boolean Expressions

3.4.1. Syntax

```
<relational operator> ::=  < | ≤ | = | ≥ | > | ≠

<relation> ::=  <arithmetic expression> <relational operator>
   <arithmetic expression> | <logical expression> <relational operator>
                                                   <logical expression>

<Boolean primary> ::= <logical value> | <variable> |
   <function designator> | <relation> | (<Boolean expression>)

<Boolean secondary> ::=  <Boolean primary> | ¬<Boolean primary>

<Boolean factor> ::=  <Boolean secondary> |
   <Boolean factor> ∧ <Boolean secondary>

<Boolean term> ::=  <Boolean factor> | <Boolean term>
   ∨ <Boolean factor>

<implication> ::=  <Boolean term | <implication) ⊃ <Boolean term>

<simple Boolean) ::=  <implication> |
   <simple Boolean> ≡ <implication>

<Boolean expression> ::=  <simple Boolean> |
   <if clause> <simple Boolean> else <Boolean expression>
```

3.4.2. Examples

$x = -2$

$Y > V \vee z < q$

$a + b > -5 \wedge z - d > q \uparrow 2$

$p \wedge q \vee x \neq y$

$g \equiv \neg a \wedge b \wedge \neg c \vee d \vee e \supset \neg f$

$\underline{\text{if}}\ k < 1\ \underline{\text{then}}\ s > w\ \underline{\text{else}}\ h \leq c$

$\underline{\text{if}}\ \underline{\text{if}}\ \underline{\text{if}}\ a\ \underline{\text{then}}\ b\ \underline{\text{else}}\ c\ \underline{\text{then}}\ d\ \underline{\text{else}}\ f\ \underline{\text{then}}\ g\ \underline{\text{else}}\ h<k$

3.4.3. Semantics

A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4. Types

Variables and function designators entered as Boolean primaries must be declared <u>Boolean</u> (cf. section 5.1. Type Declarations and sections 5.4.4. Values of Function Designators).

3.4.5. The Operators

Relations take on the value <u>true</u> whenever the corresponding relation is satisfied for the expressions involved, otherwise <u>false</u>.

The meaning of the logical operators $\neg$, (not), $\wedge$ (and), $\vee$ (or), $\supset$ (implies), and $\equiv$ (equivalent), is given by the following function table.

b1	<u>false</u>	<u>false</u>	<u>true</u>	<u>true</u>
b2	<u>false</u>	<u>true</u>	<u>false</u>	<u>true</u>
$\neg$ b1	<u>true</u>	<u>true</u>	<u>false</u>	<u>false</u>
b1$\wedge$b2	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>
b1$\vee$b2	<u>false</u>	<u>true</u>	<u>true</u>	<u>true</u>
b1$\supset$b2	<u>true</u>	<u>true</u>	<u>false</u>	<u>true</u>
b1$\equiv$b2	<u>true</u>	<u>false</u>	<u>false</u>	<u>true</u>

3.5. Logical Expressions

3.5.1. Syntax

<simple logical operator> ::= $\vee$ | $\wedge$ | $\neg$ | $

<logical primary> ::= <octal integer> | <integer> | <variable> |
  <function designator> | <Boolean expression> | (<logical expression>)

<shift measure> ::= <arithmetic expression>

<logical secondary> ::= <logical primary> | $\neg$<logical primary>

<logical factor> ::= <logical secondary> | <logical factor> | $ <shift measure> |

<logical term> ::= <logical factor> | <logical term> $\wedge$ <logical factor>

<major> ::= <logical term> | <major> $\vee$ <logical term>

<logical expression> ::= <major> | <if clause> <major> <u>else</u> <logical expression>

3.5.2. Examples

X $\vee$ Y

X $ (I * J - 4)

(X $\wedge$ Y) $ K

(<u>if</u> X $ 2 = Y <u>then</u> Z else K) $ 3

3.5.3. Semantics

A logical expression is a rule for computing the value of a fixed length string of binary digits. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.5.4. Types

Variables and function designators entered as logical primaries must be declared <u>logical</u> (cf. section 5.1. Type Declarations and sections 5.4.4. Values of Function Designators).

3.5.5. The Operators

The operator $ refers to a (non-cyclic) shift of the binary pattern. Thus in $\ell 1$ \$ $\ell 2$, the logical variable $\ell 1$ is shifted $|\ell 2|$ (mod σ) places left (right) if $\ell 2$ has a positive (negative) value. σ is a function of the register size of a computer and will, of course, vary among computers.

All other operators are as described in sections 3.4.5. and 3.3.4.

3.5.6. Precedence of operators

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.5.6.1. According to the syntax given in section 3.4.1, 3.5.1, the following rules of precedence hold:

first: arithmetic expressions according to section 3.3.5.

second: $< \leqq = \geqq > \neq$

third: $\neg$

fourth: $

fifth: $\wedge$

sixth: $\vee$

seventh: $\supset$

eighth: $\equiv$

3.4.6.2. The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.6. <u>Designational Expressions</u>

3.6.1. Syntax

```
<label> ::= <identifier> | <unsigned integer>
<switch identifier> ::= <identifier>
<switch designator> ::= <switch identifier> [<subscript expression>]
<simple designational expression> ::= <label> | <switch designator> |
        (<designational expression>)
```

continued --

<designational expression> ::= <simple designational expression> |
    <if clause> <simple designational expression> <u>else</u>
    <designational expression>

3.6.2. Examples

17

p^9

Choose[n-1]

Town[<u>if</u> y<0 <u>then</u> N <u>else</u> N+1

<u>if</u> Ab<c <u>then</u> 17 <u>else</u> q[<u>if</u> w≤0 <u>then</u> 2 <u>else</u> n]

3.6.3. Semantics

A designational expression is a rule for obtaining a label of a statement (cf. section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the <u>if</u> clauses will select a simple designational expression. If this is a label the desired result is already found. A <u>switch</u> designator refers to the corresponding <u>switch</u> declaration (cf. section 5.3. Switch Declarations) and by the actual numbrical value of its subscript expression selects one of the designational expressions listed in the <u>switch</u> declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.6.4. The subscript expression

The evaluation of the subscript expression is analogous to that of subscripted variables (cf. section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1, 2, 3, ..., n, where n is the number of entries in the switch list.

3.7 Address expressions

3.7.1 Syntax

```
<address expression> ::= |<indirect> |→<direct>
<indirect> ::=  <indirect address expression> |(<indirect address
    expression> <sign> <indirect address expression>) |(<indirect>)
<direct> ::=  <unsigned address integer> |<indirect>
<indirect address expression> ::=  <simple address expression>|
     (<indirect address expression>)
<simple address expression> ::= <elementary address> |<elementary address>
              <sign> <elementary address> |<elementary address>
              <sign> <unsigned address integer>
<elementary address> ::=  <identifier>
<unsigned address integer> ::= <an unsigned integer ≤ 32,767>
<sign> ::=  + | -
```

3.7.2. Examples

```
  |A 26
  |B + (8)23
 →C -  46 + DT4
 →D1 + E4
  |D2 + ( E9 )
 →( K6 + (Z I 11))
  |((( ALPHA ). + ( MU )) + (TAU))
```

3.7.3. Semantics

Address expressions are used to specify the value of operands of the symbolic machine-like code. Their syntax is defined to make maximal use of the operand generating facilities of a particular computer. In particular, in address expressions, the characters "("and ")" bracketing an identifier refer to the contents of the storage location which will correspond to that identifier. Nested parentheses provide levels of indirect addressing. | indicates that identifiers not enclosed in parentheses have individually implied parentheses about them. → indicates that the value of the address expression is the direct operand . The value of the address expression will in general be defined modulo (∫) where ∫ will depend on the addressible storage capacity of the computer.

3.7.4. Precedence of operators

The sequence of operations within an address expression is generally from left to right. Insofar as sequencing of operations is concerned, the use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.8 List Subscript expressions

3.8.1. Syntax

```
<address chain> ::= ↙|↘|↙<address chain>|↘<address chain>|
                                      <function designator>|<empty>
<forelist subscript expression> ::= p|<address chain> |p <address chain>
<aftlist sequence chain head> ::= |*|<function designator>|
<aftlist sequence chain> ::= <aftlist sequence chain head>|<aftlist
                           sequence chain head> <aftlist sequence chain>
<aftlist subscript expression> ::= ¢ |<aftlist sequence chain>
```

3.8.2. Examples: (as subscripts)

```
L [↙↙↘, ¢ ]
L [ p↙, * ]
L [ p  H(V[↙ *], t)↘, * * ]
```

3.8.3. Semantics

The list subscript expressions are used to select list components. p isolates the list component prefix, ↙(↘) the left (right) portion of the list component. ¢ refers to the current position of the sequence counter on the list in question; * refers to the next as defined by the sequencing rule invoked on the list. # alters the sequence counter before extraction of the list component.

3.8.4. Precedence of operators

↘and ↙ are associative to the right, i.e.,

↙↙↙x means ↙of ↙ of ↙ of x.

4. Statements

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by control statements, i.e., go to statements, which define their successor explicitly; shortened by conditional statements, which may cause certain statements to be skipped; and expanded by for statements which cause certain statements to be repeated.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1. Compound Statements and Blocks

4.1.1. Syntax

<unlabelled basic statement> ::= <assignment statement> | <go to statement> | <dummy statement> | <procedure statement> | <code line>

<basic statement> ::= <unlabelled basic statement> | <label>: <basic statement>

<unconditional statement> ::= <basic statement> | <for statement> | <compound statement> | <block>

<statement> ::= <unconditional statement> | <conditional statement>

<compound tail> ::= <statement> end | <statement> ; <compound tail>

<block head> ::= begin <declaration> | <block head> ; <declaration>

<unlabelled compound> ::= begin <compound tail>

<unlabelled block> ::= <block head> ; <compound tail>

<compound statement> ::= <unlabeled compound> | <label>:<compound statement>

<block> ::= <unlabelled block> | <label>:<block>

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

L: L: ... begin S ; S ; ... S ; S end

Block:

L: L: ... begin D ; D ; .. D ; S ; S ; ...S ; S end

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

4.1.2. Examples

Basic statements:

```
a := p+q

go to Naples

START: CONTINUE: W:= 7.993

r : b :   CIA   X + (B) ↓
```

Compound statement:

```
begin x := 0  ;  for y :=  1 step 1 until n do x := x+A[y]  ;
      if x>q then go to STOP else if x>w-2 then go to S  ;
      Aw: St: W:= x+bob end
```

Block:

```
Q: begin integer i, k  ;  real w   ;
     for i :=  1 step 1 until  m  do
     for k :=  i + 1 step 1 until  m do
     begin w :=  A[i,k]  ;
        A[i,k] := A[k,i]  ;
        A[k,i] := w end for i and k
   end block Q
```

4.1.3. Semantics

Every block automatically introduces a new level of nomenclature. This is realised as follows: Any identifier occurring within the block may through a suitable declaration (c.f. section 5. Declarations) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be nonlocal to it, i.e., will represent the same entity inside the block and in the level immediately outside it. The exception to this rule is presented by labels, which are local to the block in which they occur.

Since a statement of a block may again itself be a block the concepts local and nonlocal to a block must be understood recursively. Thus an identifier, which is nonlocal to a block A, may or may not be nonlocal to the block B in which A is one statement.

4.2. Assignment Statements

4.2.1. Syntax

```
<left part> ::= <variable> :=
<left part list> ::= <left part> |<left part list> <left part>
<assignment statement> ::= <left part list> <arithmetic expression>|
    <left part list> <Boolean expression>
```

4.2.2. Examples

$s := p[0] := n := n + 1 + s$

$n := n + 1$

$A := B/C - v - q \times S$

$S[v, k+2] := 3 - \mathrm{arctan}(s \times zeta)$

$V := Q > Y \wedge Z$

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables. The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4. Types

All variables of a left part list must be of the same declared type. If the variables are Boolean, the expression must likewise be Boolean. If the variables are of type <u>real</u> or <u>integer</u>, the expression must be arithmetic. If the type of the arithmetic expression differs from that of the variables, appropriate transfer functions are understood to be automatically invoked. For transfer from <u>real</u> to <u>integer</u> type, the transfer function is understood to yield a result equivalent to

entier(E + 0.5)

where E is the value of the expression.

4.3. <u>GO</u> <u>TO</u> Statements

4.3.1. Syntax

<go to statement> ::= <u>go</u> <u>to</u> <designational expression>

4.3.2. Examples

<u>go</u> <u>to</u> 8

<u>go</u> <u>to</u> exit [n+1]

<u>go</u> <u>to</u> Town[<u>if</u> y<0 <u>then</u> N <u>else</u> N+1]

<u>go</u> <u>to</u> <u>if</u> Ab< c <u>then</u> 17 <u>else</u> q[if w<0 <u>then</u> 2 <u>else</u> n]

4.3.3. Semantics

A <u>go</u> <u>to</u> statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4. Restriction

Since labels are inherently local, no go to statement can lead from outside into a block.

4.3.5. GO TO an undefined switch designator

A go to statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.4. Dummy Statements

4.4.1. Syntax

<dummy statement> ::= <empty>

4.4.2. Examples

L:

begin ...   John: end

4.4.3. Semantic

A dummy statement executes no operation. It may serve to place a label.

4.5. Conditional Statements

4.5.1. Syntax

<if clause> ::= if <Boolean expression> then

<unconditional statement> ::= <basic statement> | <for statement> |
    <compound statement> | <block>

<if statement> ::= <if clause> <unconditional statement> |
    <label>:<if statement>

<conditional statement> ::= <if statement> | <if statement> else
    <statement>

4.5.2. Examples

```
    if  x>0 then n := n+1

    if  v>u then V : q: = n+m else go to R

    if  s<0∨P≤Q then AA: begin if q<v then a := v/s
                 else y := 2 × a end
               else if  v>s then a := v-q else if v>s-1
                 then go to  S
```

4.5.3. Semantics

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1. __If__ statement. The unconditional statement of an __if__ statement will be executed if the Boolean expression of the __if__ clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2. Conditional statement. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

```
    if B1 then S1 else if B2 then S2 else S3  ;  S4
```

and

```
    if B1 then S1 else if B2 then S2 else if B3 then S3  ;  S4
```

Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expression of the __if__ clauses are evaluated one after the other in sequence from left to right until one yielding the value __true__ is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, i.e., the statement following the complete conditional statement. Thus the effect of the delimiter else may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

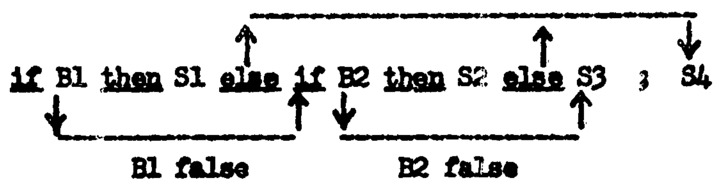
else <unconditional statement>

is equivalent to

else if true then <unconditional statement>

If none of the Boolean expressions of the if clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:



4.5.4. GO TO into a conditional statement

The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the effect of else.

4.6. For statements

4.6.1. Syntax

```
<for list element> ::= <arithmetic expression>|
    <arithmetic expression> step <arithmetic expression> until
    <arithmetic expression>|<arithmetic expression> while
    <Boolean expression>
<for list> ::= <for list element>|<for list>, <for list element>
```

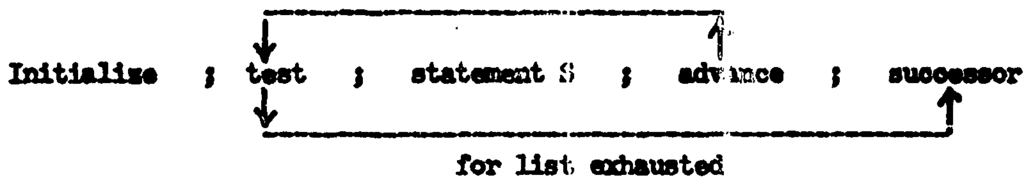
```
<for clause> ::= for <variable> := <for list> do
<for statement> ::= <for clause> <statement>|
    <label>:<for statement>
```

4.6.2. Examples

```
for  q := 1 step s until n do A[q] := B[q]
for  k := 1, V1 x 2 while V1<N do
        for j := I+G, L, 1 step 1 until N, C+D do
            A[k,j] := B[k,j]
```

4.6.3. Semantics

A for clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualised by means of the following picture:



In this picture the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment of the for clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the for statement. If not, the statement following the for clause is executed.

4.6.4. The for list elements

The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1. Arithmetic expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2. Step-until-element. A for element of the form A step B until C, are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional 20∧L statements as follows:

```
      V  :=  A    ;
L1 : if (V-C) x sign(B)>0 then go to Element exhausted;
      Statement S  ;
      V := V+B    ;
      go to L1    ;
```

where V is the controlled variable of the for clause and Element exhausted points to the evaluation according to the next element in the for list, or if the step-until-element is the last of the list, to the next statement in the program.

4.6.4.3. While--element. The execution governed by a for list element of the form E while F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

```
L3 : V := E ;
    if ¬ F then go to Element exhausted ;
    Statement S ;
    go to L3 ;
```

where the notation is the same as in 4.6.4.2 above.

4.6.5. The value of the controlled variable upon exit.

Upon exit out of the statement S (supposed to be compound) through a go to statement the value of the controlled variable will be the same as it was immediately preceding the execution of the go to statement.

If the exit is due to exhaustion of the for list, on the other hand, the value of the controlled variable is undefined after the exit.

4.6.6. Go to leading into a for statement

The effect of a go to statement, outside a for statement, is undefined.

4.7. Procedure Statements

4.7.1. Syntax

```
<actual parameter> ::= <string> | <expression> | <array identifier> |
    <list identifier> | <switch identifier> | <procedure identifier>
<letter string> ::= <letter> | <letter string> <letter>
<parameter delimiter> ::= , | <letter string>:(
<actual parameter list> ::= <actual parameter> |
    <actual parameter list> <parameter delimiter>
    <actual parameter>
<actual parameter part> ::= <empty> |
    (<actual parameter list>)
<procedure statement> ::= <procedure identifier>
    <actual parameter part>
```

4.7.2. Examples

```
Spur  (A) Order: (7)Result to: (V)
Transpose (W,v+1)
Absmax (A,N.M,Yy,I,K)
Innerproduct(A[t,P,u],B[P],10,P,Y)
Among (V,W[ y≠ ])
```

These examples correspond to examples given in section 5.4.2.

4.7.3. Semantics

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. Procedure Declarations). Where the procedure body is a statement written in 20ΛL the effect of this execution will be equivalent to the effect of performing the following operations on the program:

4.7.3.1. Value assignment (call by value)

All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and Types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. These formal parameters will subsequently be treated as local to the procedure body.

4.7.3.2. Name replacement (call by name)

Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3. Body replacement and execution

Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed.

4.7.4. Actual-formal correspondence

The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5. Restrictions

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2. lead to a correct 20^L statement.

This imposes the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. Strings cannot occur as actual parameters in procedure statements calling procedure declarations having 20^L 60 statements as their bodies (c.f. section 4.7.8).

4.7.5.2. A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression].

4.7.8. Procedure body expressed in code

The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

4.8. Code Line

4.8.1. Syntax

```
<code line> ::= <line of code>↓
<line of code> ::= <instruction designator> |<macro designator>|<empty>
<instruction designator> ::= <operation designator> <address expression>
<operation designator> ::= <any of the operation mnemonics of the
                                  computer machine code>
<macro designator> ::=  <macro identifier> <actual parameter part>
```

4.8.2. Examples

```
CLA → X + 4↓
STF |(X + (B) )↓
GRP |( Z, ( (A 9) - ( h 13 ) ), 7 )↓
```

4.8.3. Semantics

The code line is the unit statement in machine-like symbolic code. In the case of macro designators, the parenthesis conventions of the actual parameters must match the requirements of the formal parameters in the corresponding macro declaration

4.9. Macro Statements

4.9.9. Syntax

```
<actual elementary macro parameter> ::= <string not containing parentheses
       <a.e.m.p.>                                                     or ,>

<actual macro parameter> ::=  <a.e.m.p.> | (<a.m.p.l.>)
       <a.m.p.>

<actual macro-parameter list> ::=  <a.m.p.> | <a.m.p.l.> , <a.m.p.>
       <a.m.p.l>

<actual macro parameter part> ::= <empty> | (<a.m.p.l.>)

<macro statement> ::= <macro identifier> | <actual macro parameter part>
```

4.9.2. Examples

```
Innerproduct ( ACt, P, u], B[P], 10, P, Y )

Among   ( (R∧T) $ 1, W, C)
```

4.9.3. Semantics

A macro designator specifies that the following sequence of events transpire:

(i) In a copy of the macro declaration corresponding to the macro designator, the set of characters specifying an actual parameter are substituted for their corresponding formal parameters in all places of the latter's occurrence in the macro declaration. Then

(ii) The altered (copy of the) declaration replaces the macro statement which called it and then

(iii) Processing continues at the code position previously occupied by the macro statement.

5. Declarations

Declarations serve to define certain properties of the identifiers of the program. A declaration for an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3).

Dynamically this implies the following: at the time of an entry into a block (through the begin, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block(through end. or by a go to statement) all identifiers which are declared for the block lose their significance again.

A declaration may be marked with the additional declarator own. This has the following effect: upon a reentry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax.

<declaration> ::= <type declaration> | <array declaration> | <switch declaration> | <procedure declaration> | <macro declaration> | <parameter declaration> | <equivalence declaration> | <library declaration> | <constant declaration>

5.1. Type Declarations

5.1.1. Syntax

<type list> ::= <simple variable> | <simple variable>, <type list>

<type> ::= real | integer | Boolean | index | list |

<local or own type> ::= <type> | own <type>

<type declaration> ::= <local or own type> <type list>

5.1.2. Examples

integer p,q,s

own Boolean Acryl,n

logical Student, Z7

5.1.3. Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero, and be represented in either decimal or octal form. Boolean declared variables may only assume the values true and false. Logical declared variables are binary strings. List declared variables are empty lists containing one information site. Index declared variables are index registers.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of own, see the fourth paragraph of section 5 above.

5.2. Array Declarations

5.2.1. Syntax

```
<lower bound> ::= <arithmetic expression>
<upper bound> ::= <arithmetic expression>
<bound pair> ::=  <lower bound>:<upper bound>
<bound pair list> ::=  <bound pair> | <bound pair list>, <bound pair>
<array segment> ::=  <array identifier> [<bound pair list>]
     <array identifier>, <array segment>
<array list> ::= <array segment> | <array list>, <array segment>
<array declaration> ::=  array <array list> | <local or own type>
     array <array list>
```

5.2.2. Examples

```
array  a, b, c[7:n,2:m], s[-2:10]
own integer array  A[if c<0 then 2 else 1:20]
real array q[-7:-1]
```

5.2.3. Semantics

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables.

5.2.3.1. Subscript bounds. The subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter: The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions. The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3. Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type <u>real</u> is understood.

5.2.4. Lower upper bound expressions

5.2.4.1. The expressions will be evaluated in the same way as subscript expressions (cf. section 3.1.4.2).

5.2.4.2. The expressions can only depend on variables and procedures which are non-local to the block for which the array declaration is valid. Consequently in the outermost block of a program only <u>array</u> declarations with constant bounds may be declared.

5.2.4.3. An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4. The expressions will be evaluated once at each entrance into the block.

5.2.5. The identity of subscripted variables

The identity of a subscripted variable is not related to the subscript bounds given in the <u>array</u> declaration. However, even if an array is declared <u>own</u> the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3. <u>Switch Declarations</u>

5.3.1. Syntax

```
<switch list> ::= <designational expression> |
        <switch list>, <designational expression>
<switch declaration> ::= switch <switch identifier> := <switch list>
```

5.3.2. Examples

```
switch S := S1,S2,Q[m], if v>-then S3 else S4
switch Q := p1,w
```

5.3.3. Semantics

A switch declaration defines the values corresponding to a switch identifier. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, ..., obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.6. Designational Expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4. Evaluation of expressions in the switch list

An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5. Influence of scopes.

Any reference to the value of a switch designator from outside the scope of any quantity entering into the designational expression for this particular value is undefined.

5.4. Procedure Declarations

5.4.1. Syntax

```
<formal parameter> ::= <identifier>
<formal parameter list> ::= <formal parameter> |
    <formal parameter list> <parameter delimiter>
    <formal parameter>
<formal parameter part> ::= <empty> | (<formal parameter list>)
<identifier list> ::= <identifier> | <identifier list>,<identifier>
<value part> ::= value<identifier list> ; | <empty>
<specifier> ::= string | <type> | array | <type> array | label | switch |
    procedure | <type>procedure | list
<specification part> ::= <empty> | <specifier> <identifier list> ; |
    <specification part> <specifier> <identifier list> ;
<procedure heading> ::= <procedure identifier>
    <formal parameter part> ; <specification part> <value part>
<procedure body> ::= <statement> | <code>
<procedure declaration> ::=
    procedure <procedure heading> <procedure body> |
    <type> procedure <procedure heading> <procedure body>
```

5.4.2. Examples (see also the examples at the end of the report).

```
procedure Spur(a)Order:(n)Result:(s) ; value n ;
array a ; integer n ; real s ;
begin integer k ;
s := 0 ;
for k := 1 step 1 until n do s := s + a[k,k]
end
```

Examples continued:

```
procedure Transpose(a)Order:(n)  ;  value n  ;
array a  ;  integer n  ;
begin real w   ;  integer i, k   ;
for i := 1 step 1 until n do
     for k := 1+i step 1 until n do
     begin w := a[i,k]   ;
          a[i,k] := a[k,i]  ,
          a[k,i] := w
     end
end Transpose

integer procedure Step(u)   ;   real u   ;
Step := if 0≦u∧u≦1 then 1 else 0

procedure Absmax(a)size:(n,m)Result:(y)Subscripts:
     (i,k)   ;
comment The absolute greatest element of the matrix a,
     of size n by m is transferred to y, and the subscripts
     of this element to i and k   ;
array a   ; integer n, m, i, k   ;   real y   ;
begin integer p, q   ;
y := 0   ;
for p := 1 step 1 until n do for q := 1 step 1 until m do
if abs(a[p,q])> y then begin y:=abs(a[p,q])   ;  i:=p   ;
     k:=q
end end Absmax
```

Examples continued:

```
        procedure  Innerproduct(a,b)Order:(k,p)Result:(y)   ;
             value k   ;
        integer k,p   ;   real y,a,b,   ;
        begin real s   ;
        s:=0  ;
        for p := 1 step 1 until k do s :=s+a x b   ;
        y := s
        end  Innerproduct
```

5.4.3. Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. section 3.2. Function Designators and section 4.7. Procedure Statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are nonlocal to the body may well be local to the block in the head of which the procedure declaration appears.

5.4.4. Values of function designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur an assignment of a value to the procedure identifier, and in addition the type of this value must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration.

Any other occurrence of the procedure identifier within the procedure body denotes activation of the procedure.

5.4.5. Specifications

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once and formal parameters called by name (cf. section 4.7.3.2) may be omitted altogether.

5.4.6. Code as procedure body

It is understood that the procedure body may be expressed in non-20^L language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language.

5.5. Macro Declarations

5.5.1. Syntax

<macro heading> ::= <macro identifier> <formal parameter part> ;
<specification part>

<macro body> ::= <statement>

<macro declaration> ::= **macro** <macro heading> <macro body>

5.5.2. Examples

```
macro    Innerproduct (a,b) Order:(k,p)Result:(y)    ;
integer k,p ; real S  ;
  S: = 0
for p: = |step|  until k do  S: = S + a x b  ;
  y: = S end Innerproduct
macro  Among (x,y) Predicate:(B) Exit:(L)
  list  y  ;
  Boolean B  ;
  logical  x  ;
  comment B is true if and only if the logical variable  x  is
an (indirect) element of the list y ;
begin  seqe(y,L) ; B: = false  ;
  S: If  y[p,*] $-1 ≠ 0 ∧ y[ ⅇ,ø] = x  Then begin B:= true  ;
                                                go to L end .
else  go to S
end  Among.
```

5.5.3. Semantics

A macro declaration serves to define a macro associated with a macro identifier. Macros only exist in the processing interval from their point of definition in the lexicographic sequencing of code to the end of the block in which they are defined — and, in time, only during Pass 1.

The principal constituent of a macro declaration is a statement. Associated with the body is a heading, much as with procedures, except the concept of name, value have no significance with macros. Whenever a macro is called, the formal parameters in the macro body will be replaced by the actual parameters corresponding leading to a new section of 20^L code which will then be immediately subject to processing. Replacement is understood to occur simultaneously on all parameters -- all their occurrences in the macro body.

5.5.4. The replacement process

Any string of characters satisfying the syntactic rules of actual macro parameters, (see section 4.9) may replace an identifier.

5.6. Equivalent declaration

5.6.1. Syntax

```
<inner element> ::= <simple variable> |
<outer element> ::= <simple variable> | <subscripted variable>
<simple pair> ::= (<inner element>, <outer element>) |
<pair list> ::= <simple pair> | <pair list>, <simple pair>
<equivalent declaration> ::= equivalent (<pair list>
```

5.6.2. Examples

<u>equivalent</u> (A,B), (TAU, PHI [i,j+1])

5.6.3. Semantics

An equivalent declaration declares an identifier, (the inner element) within the block containing the equivalence declaration to be--in every respect--identical with an identifier(the outer element) declared in an outer block.

5.7. Library declaration

5.7.1. Syntax

<identifier list> ::= <identifier> | <identifier list>,<identifier>

<library declaration head> ::= <local or own type> | <empty>

<library declaration title> ::= <u>library</u> | <u>library</u> <u>procedure</u>

<library declaration> ::= <library declaration head> <library declaration title> <identifier list>

5.7.2. Examples

<u>library</u> RANDOM, NTHROOT

<u>integer</u> <u>library</u> SORT

<u>library</u> <u>procedure</u> X, MTXINVSE

5.7.3. Semantics

The library declaration serves to call a machine coded, non 20^L, procedure from the library. Connection to the procedure is made by a standard procedure statement. All storage requirements, except actual parameters, are provided within the library.

The <u>library</u> <u>procedure</u> declaration calls a 20^L procedure declaration from the library to be substituted in the 20^L code for the occurrence of the library procedure declaration. Substitution of these procedure declarations is made in the left to right order of their names in the call.

Part II Flow Charts for $20^{\wedge}P$.

1. Philosophy

Flow charting an operation of a processor such as $20^{\wedge}P$ must inevitably be complex and--at times--somewhat machine dependent. Every effort has been made to minimize references which are of the latter kind. Nevertheless, where they are required, specific machine instructions will be used and their explanation given at that time. The flow charts will be given at several levels of description and the detail at any given level willbe a function of the processes being described. Some descriptions will be in text, others in a semi-formal notation.

Flow charts are extremely difficult to read under the best of circumstances and the use of formal notation exclusively makes them elegant but impossible to comprehend. However, occasionally it serves the admirable purpose of a shorthand notation and it will be used in such places and defined at the point of use.

As has been mentioned the processor operated in three phases: P1, P2, P3, loosely described as passes over the code.

P1 is an assembly phase and a translation phase

P2 is a compiling and loading phase.

P3 is a running or operating phase.

Intervening are two transition phases T1, T2, which work on tables prepared during passes P1, P2 ; and P2 P3, respectively. During these phases for many of the $20^{\wedge}L$ elements there is a point of declaration (D) and (often several) subsequent points of call or use (C). The notation P1,D will refer to some action taken at a declaration during pass 1.

2. Pass 3 Disposition

The processing becomes more clear when a storage map of a 20^L program is specified. Schematically it is:

		Flow Chart designation
Fixed Data Storage	own arrays	FO
	fixed arrays	FA
	constants	FC
	scalar variables	FS
	list table	FL
Library		L
and		
Administration		BA
List Storage Pool		LSP
Program		P
Dynamic arrays		DA
and		
Scalars		DS
Parameter stack		PS

Distributed through this storage certain tables are present in all or most programs. They are:

1.	The list table	LT
2.	The active block or porcedure table	ABT
3.	The parameter stack	PS
4.	The exit stack	ES

During P1,D as a block is encountered (<u>begin</u> <u>declaration</u> ...) it is assigned a name (Br), a level (Bs), and a tag (Bt). The names are positive integers satisfying: If the block <u>begin</u> of a block α occurs (lexicographically) before the block <u>begin</u> of a block B then $Br_{\alpha} < Br_{B}$. The levels specify the depth of parenthesis nesting with <u>begin</u> an opening parenthesis and <u>end</u> a closing one. Each level Bℓ will correspond to an assigned Index register. The block tag (Bt) specifies whether the block is internal to a procedure(Bt = 1) or not(Bt = 0). The term 1-block or 0-block will be used to distinguish blocks by this property. 1-blocks can, of course, be part of a recursive process and its declared variables must be treated dynamically.

In the case of 1-blocks, P1,D must generate code operative during P3,D which supplies information to the Administration routine BA. The information supplied must carry to BA:

(i) The block name

(ii) The block level

This information is added to ABT in an augmented table line:

Block name	Block level	Base address
r	s	Ar

A_r had been generated by ABT and then index register S is loaded with Ar.

In either case, for 1-blocks and 0-blocks, a code line is entered into the Code Block Table (C B T) active during P1,D:

line number	Bt	Block name	Block level	Base address
7	g	r	s	␣

3. Code generation and Table Generation

During P1 and P2 various code and table generation actions take place. If a table has the name $\textcircled{H}$, then $\textcircled{H}\not{c}$ indicates the line at which a marker is poised. $\textcircled{H}$ * indicates that the marker has been moved forward one line. If $\textcircled{H}$ has a field structure, substitution into and extraction from fields is indicated by a variety of obvious notations. Thus, e.g.,

$$\text{CBT}\,* \leftarrow ,,,\text{Bt},,\text{Br},\text{Bs},\,\sqcup$$

specifies that the contents of Bt, Br, and Bs are put into the 3, 5, and 6th field of the next line of CBT and $\sqcup$ (blank) into the 7th field.

During P1, codo generation occurs. This code is either $20^\wedge$L code (say, as a result of macro calls) or a form of 3 address code, written, e.g.,

$$\Upsilon 3.\ \theta.\Upsilon 2.\Upsilon 1$$

meaning $\Upsilon 3 \leftarrow \Upsilon 2\ \theta\ \Upsilon 1$ in conventional notation. In a given line any of $\Upsilon 1$, $\Upsilon 2$, $\Upsilon 3$ may be missing. A collection of 3-address code will be recognised as being collected as a unit by the notation

$$(,\xi_1,\ \xi_2,\ \ldots,\ \xi_k)$$

where each ξ_i is such a 3-address code line.

The operators θ will be those most immediately derived from the semantics of

The operators θ will be those most immediately derived from the semantics of 20^L operators. A table containing the notation and meaning or all such follows:

3-address operator table

arithmetic	
+	addition
x	multiplication
/	division
÷	integer division
↑	exponentiation
π	store in parameter stack
MT	mark transfer
σ	store
Ph	procedure begin
Pe	procedure end
Bh	block begin
Be	block end
L	label

In P2 the three address code is converted into machine code. Here, detailed knowledge of a machine is necessary and only the general method of such translating can be discussed without becoming overinvolved in specific computer details.

In P1, where the major assembly and translation functions are accomplished there are two major code sequences:

(i) 20$^\wedge$L (input) code

(ii) List organized three-address code.

The organisation of these two code sequences is quite different. (i) is organized as a linear chain while (ii) is organised as a tree. The fundamental distinction is the mode of their sequencing. Each has a marker denoted by a sub-line $\uparrow$; and this marker moves through the sequences in different ways. To be specific:

In the case of

(i) The marker may move relative to its current position designated as I ¢ (Input-current)--any number of character positions (spaces excluded) backwards or forwards.

Whereas

(ii) The marker may only move back by moving forward to a list sentinel and up several lists higher in the tree structure, through the application of the sequence procedure "up". On the other hand the code position θ ¢ may itself have substituted into it an entire list through which the marker position may pass or by-pass as occasion demands.

In the case of Pl the scanning of the input sequence (IS) is the clocking mechanism: The number of characters to the right of the marker is non-decreasing.

The scanner is quite elementary. Characters are either components of identifiers, numbers, truth values, or delimiters. In the case of identifiers and numbers, these syntactic units are multi-charactered, and, when encountered, are accumulated in an accumulator until a delimiter is reached. In this context this delimiter is known as a terminal element.

The scanner is under control of Pl translator routines each of which has a set of terminal elements, the occurrence of which create the conditions under which these routines select their actions.

Of course, these terminal elements may themselves serve to activate translator routines.

In order to organize the translation scheme, the $\mathcal{O}$S code itself is the master control scheme. Sites in the $\mathcal{O}$ code may contain the names of translator routines, e.g., X, designated $\{X\}$, or, of course, 3-address code. Progressing through the $\mathcal{O}$ code now specifies the control organization of the translation process.

As a matter of efficiency, certain translator routines have their own sub-control organization; for example, the "expression" translator, and that for some of the declarations. Indeed, the "expression" translator produces 3-address code in a block rather than a list.

In the case of many of these special routines, the sub-routines function when certain character sequences occur in IS. Their occurrence causes certain actions, these are combinations of:

(i) Code generation

(ii) Substitution into, and movement of the markers in, IC and $\mathcal{O}$C

(iii) Operations on the auxiliary tables generated during the Pass.

These actions will be noted in the form of productions:

$$\beta: \quad S_1 \vec{\mathcal{F}} S_2 \Rightarrow a_1, a_2, \ldots, a_k \;;\; \underline{\text{go}}\ \underline{\text{to}}\ \alpha \mid \underline{\text{go}}\ \underline{\text{to}}\ \gamma$$

meaning: If the character string S_1 is of the form $(\vec{\mathcal{F}})$ S_2 then actions $a_1, a_2, \ldots, a_k$ are accomplished; following which, α is the label of the next production accomplished. If not of the form, γ is the label of the next production.

4. The flow charts proper.

The analysis of a $20^{\wedge}$L program is controlled by the block structure of $20^{\wedge}$P. Thus the flow charts naturally divide into:

(i) The analysis of declarations since they define that which occurs at the beginning of a block.

(ii) The analysis of statements since they form the content of a block.

(iii) The analysis of expressions since they form the content of most statements.

(iv) The analysis of identifiers since they form the content of most expressions

(v) The analysis ofsthe block end since administration of storage and identifier scopes is controlled in that way.

The flow charts reduce $20^{\wedge}$L in a 3 address pseudo code plus certain tables out of which a machine dependent Pass 2 would produce machine code.

The 3 address code so produced is itself tied together in a list structure. In general, the notation of the charts will be that of $20^{\wedge}$L.

5. Flow charts for declarations.

```
Block:  If  I[ ,*] = ' begin  then
                        begin isrt( O[,¢]);list(O[,*]); I[,*]
                              go to  main declaration  end
                              else
            If I[,¢] = 'Blend'  then
                       begin  go to Block end end
                               else  go to statement
```

```
        comment The next step is to check the occurrence, if any, of
    a declaration;
        Main declaration: macro declare (U,V) label V;
                               begin if I[ ,¢] = U
                                   then
                                 begin MT Blockhead; go to V end end
                          declare( 'own',own1)
        real:             declare( 'real, real 1)
                          declare ( 'integer', integer 1)
                          declare ( 'logical', logical 1)
                          declare ( 'index', index 1)
                          declare ( 'list' , list 1)
                          declare ( 'Boolean', Boolean 1)
        array:            declare ( 'array',  array 1)
                          declare ( 'procedure', procedure 1)
                          declare ( 'library', library 1)
                          declare( 'equivalent', equivalent 1)
                          declare( 'macro', macro 1)
                          If I[,¢] ='comment' then go to comment
                          If I[,¢] ='value' then go to value
                          go to compound
    comment The preceding is the switching table for declarations in 20^L
        Blockhead:        ; s:= s+ 1; r := r + 1; crblk := r;
                            field( 2, IT [ k ] ) := true
                            BT [,*] :=  Bt, | (r),(s)
    Blockhead 1 :       If  Bt
                        then
              begin     isrt ( 3, O[,¢])
                        O[,*] := code line ( ' _. Bh .| (r) . (s)')
                        O[,*] := code line ('_ . MT | (BA) ) end
                        go_to | (Blockhead)
comment 1.  crblk holds the index of the current block being processed
        2.  field gains access to the fields of tables
        3.  Bt is a block flag indicating whether the block is interior
            to a procedure declaration
        4.  code line is a procedure generating its actual parameter
            as a code line
        5.  BA is a fixed location whose contents specify the variable
            location of the block administration routine;
    own 1:  delta [1] := true; I[,¢] := 'dolar'; I[,*]; go_to real
    real 1: delta [2] := true; real 2:  I[,¢] :='dolar'; I[,*]
            go_to array
    integer 1:  delta [3] := true; go_to real 2
    logical 1:  delta [4] := true; go_to real 2
    Boolean 1:  delta [5] := true; go_to real 2
    index 1:  delta [6] := true; index 2:  I[,¢] := 'dolar'; I[,*]
              go_to declaration;
    list 1:  delta [7] := true; go_to index 2
    array 1 :  delta [8] := true; go_to_index 2
    procedure 1: delta [9] := true; Bt := true; I[,*]; procnest := procnest + 1
                        isrt ( O[,¢]); list ( O[,*]); isrt (2, O[,¢]);
                        param := 1; If I[,¢] = letter
                        then
                        then
                  begin A := identifier accumulated; indirect := true
                        delta [12] := true; MT identifier declared
                        O[,*] := code line ( e(A). Ph._._)
                  next( O[,¢] := code line ( e(A). Pe._._); I[,*]
                        If I[,¢] = '('
                        then
procedure 3:     begin  I[,*]; if I[,¢] = letter
                              then
                        begin  A := identifier accumulated; MT identifier declared
                              I[,*] ; if I[,¢] = ',' then go to procedure 3
                                      if I[,¢] = ')' then go to main declaration
                                      else go to alarm end
                              else go to alarm end
                        else go to alarm end
```

comment: The procedure name is declared and entered as having label character. Each of the actual parameters when declared is declared with indirect and param set to true to indicate their status. procnest specifies the depth of procedure nesting current.

```
declaration:  begin  j := 0
declaration 1:  If I[,¢] ≠ letter  then go to alarm 3
                else  A := identifier accumulated; j := j+1; M[j] := A
                      MT identifier declared
                If  I [,¢] = ','
                then go to declaration 2
```

```
                  else if  I[, ¢] = ';' then go to declaration 4
                                        else go to alarm 4
declaration 2:    K: = 1;  Expression terminal [k] : = ']'
                  MT    Expression Analyzer
declaration 3:    O[,*] : = code line (e (M[j]), σ , t, 1)
                  O[,*] : = code line (e (M[j]), σ, t,4)
                  if    j ≠ 1
                  then begin  j : = j-1; go to declaration 3 end
                  else if I[,*] = ','
                       then go to declaration 1
                       else if I[,¢] : = ';' then go to declaration 4
                                           else go to alarm 4
declaration 4:    for i : =1 step 1 until 12 do delta [i]:= 0
                  go to statement end end
```

Comment declaration handles lists of identifiers processing the same declaration and, in particular, handles array declarations. In case of arrays:

(1) of 2 dimensions A [m: n, r: s] there is computed

column = abs (s-r+1)

space = column * abs (n-m+1)

base = storage base - r - m * column n

storage base= storage base + space

and there is stored in the address assigned to A and its successor, base and column. The mapping function for A [i,j] is then base + j + i * column. The expression analyzer provides t, 1 as base and t, 4 as column;

```
equivalent 1:     j: = 0; I [,*]
                  if I[, ¢] ≠ '(' then go to alarm 5
    I[,*];        if I[,¢] ≠ letter then go to alarm 5
```

```
                A: = identifier accumulated; j: = j+1; M[j]:=A
                if I [,¢] ≠ ',' then go to alarm 6
                A: = identifier accumulated; j:=j+1; M[j]: =A
                if I[,¢] = '['
                then begin
equivalent 2:   K: =1; MT Expression Analyzer
                j: = j-1; O[,*]: = code line (e(M[j]), σ̄, t,1)
                A: = M[j]; indirect: = true ; MT identifier declared
equivalent 3:   if I [,¢] = ','
                then go to equivalent 1
                if I [,¢] = ';'
                then go to declaration 4
                go to alarm 4
                end
                else
                if  I [,¢] = ','
                then begin
                   j:=j+1; A:= M[j]; MT identifier declared; j:=j+1
                   A: = M[j]; MT chain; go to equivalent 3
                end
                go to alarm 4
```

comment equivalent makes identifiers within blocks identical to those declared outside. In the case of a variable equivalent to an array this equivalence is established dynamically.

```
value 1:        I [,¢]: = dclar
value 2:        I [,*]; if I [,¢] ≠ letter
```

```
                    then go_to alarm 3
                    A: = identifier accumulated; MT set value
                    if I [,¢] = ',' 
                    then go_to value 2
                    if I [,¢] = ';'
                    then go_to declaration 4
                    go_to alarm 4
set value:   ; delta [10]:= 1; MT identifier declared; go to set value
Comment  identifiers declared as values have that property inscribed in
the identifier's table.
Comment:         I [,*]; if I [,¢] = ';'
                            then go_to statement end
                            go_to comment
Macro:  I [,*]; delta [11]:= true
                 if I [,¢] ≠ letter
                 then go_to alarm 3
                 A: = identifier accumulated; MT identifier declared;
                 MT [,*] : = A;
                 I [,*]; I [,*]
Macro 1:         if I [,¢] = letter
                 then begin
                 A:=identifier accumulated; MI [,*]: = A,MT ¢
                 end
                 if I [,¢] = ','
                 then begin
                 I [,*]; go to macro 1
                 end
```

```
                    if I [,¢] ≠ ')'
                    then go to alarm 4
                    MT ¢: = Macro file; string transfer terminal: = 'end'
                    string transfer storage: = macro file
                    MT string transfer
                    Macro file: = Macro file + string transfer norm
                    go to declaration 4
```

Comment Macros are stored in some 'external' file whose index is in Macro file. MT is the table of Macros declared. MI is the table of Macro identifiers. String transfer is the table of Macro identifiers. String transfer is the name of a program which maps I [,¢], up to the first non-matching end, onto the Macro file;

```
library; delta[13] = 1; I [,*]; if I [,¢] ≠ procedure
                    then
library 1:  begin I [,*]; if I [,¢] = letter
                    then
                 begin A: = identifier accumulated; delta [12]: = 1;
                    MT  identifier declared; go to library 1  end
                    if I [,¢] = ','
                    then go to library 1
                    if I [,¢] = ';'
                    then  go to declaration 4
                    go to alarm 4   end library 1
                    else
library 2:   I [,*], if I [,¢] letter
                    then
                 begin A:= identifier accumulated;
                       J: = library table (A)
                       J: = field (2, J)
                       iort (2,I [,¢])
                 next  (I [,¢]):= 'library'
next next (I [,¢]): = procedure
        list ( I[,¢])
        copy (I[,¢],J); go to main declaration end
```

Comment library table contains entries by name and peripheral storage location. Procedures named in library declarations are called at the end of Pass 2. Those named in library procedure declarations are inserted into the code at the point of name.

6. The analysis of expressions.

The expression analyzer produces 3 address code from the analysis of of expressions. It acts through encountering delimiters. The delimiters upon which it acts are:

↑, *, /, ÷, +, −, <, ≤, >, ≠, $, ¬, ∧, ∨, ⊃, ≡, : , ', ',),),], else, then, :=, end, ; , =, ≤,

The order of their listing from left to right determines their order of execution within an expression.

I [,¢] has the usual meaning: indication of the current position in the the input sequence.

P [i] is the ith position in the as-yet-unfulfilled expression stack.

O [j] is the jth position in the output sequence which is a block. There is an O [~~/~~,¢] which points to this block.

6.1. Analysis of arrays.

Arrays may be characterized by two of their properties:

(a) They are of fixed (variable) dimension: a 0 (1)

(b) They are declared in a block exterior (interior) to a procedure b0 (1).

The catalogue of actions in the four cases are given in the following table:

for A [i, j]

	Declaration	Call
	A = 0	A = 0, b = 0
Pass 1	A := base A + 1 := column	code for $A + j + \overline{A+1} * i$
Pass 2		execution of above
	A = 0, b = 1	
Pass 1		code for $A + j + \overline{A+1} * i + \beta_s$ if local or compute β_s in B.A. routine store β_s in I. $A + j + \overline{A+1} * i + I$.
Pass 2		execution of above
	A = 1, b = 0	
Pass 1	A := base A+1 := column	code for $A + j + \overline{A+1} * i$
Pass 2	execution of above	execution of above
	a = 1, b = 1	
Pass 1	A [β_s] := base A + [β_s] := column	$A[s] + j + \overline{A+1}[s] * i$ if local or compute β_s in B.A. routine Store β_s in I_s
Pass 2	execution of above	execution of above

```
Expression analyser:  Ex 7: If  I [,¢] = 20^L delimiter
                              then
                           begin P [i] : = I [,¢]
                  Ex 3:    if  operator (P[i-2] ) ∧ preceeds (P[i-2], P[i-1]
                           then  go_to Ex 1
                           I [,*];  go_to Ex 7  end
                           If  I[,¢] = letter
                           then
                  begin    A: = identifier accumulated
                           MT identifier encountered
                           P[i]:= e (A)
                  Ex 2:    if P [i-1] = '-'∧P [i-2] = delimiter
                           then
                  begin    P [i] = $\overline{e (A)}$; i: = i+1; go_to Ex 7      end
                           if P [i-1] = 'else'
                           then
                  begin    P [i-4]:= P [i]; i:=i-3;  go to Ex 7   end
                           go to alarm 11   end
                           If I [,¢] = digit V I [,¢] = '.' V I [,¢] = '10'
                           then
                  begin    A: = number accumulated
                           MT number encountered
                           P [i]: = e (A);  go to  Ex 2  end
                           go to alarm 7
```

```
Ex 1:  i:= i-2; if arithmetic (P[i])
                then
        begin   if P [i-1] = t,l ∧ P[i+1] = t,m
                then
        begin   r: = min  (l,m); s: = max (l,m); temp [s] = true end
                else
        begin   if P[i-1] = t,l
                then r:=l
                else
                if P[i+1] = t,l
                then r: = l
                else r:= first available (temp)
        begin   r:= first available (temp); temp [r]:=false end
                ō [j]:= code line (t,r. (P[i]). (P[i-1]). (P[i+1]))
                j:= j+1; P[i-1]: t,r; P[i]: = P[i+2];
                go to  Ex 7   end
                if relational (P[i])
                then
        begin   if P[i-1]= t,l  then temp [l]: = true
                if P[i+1] = t,l  then temp [l]: true
                ō [j]: = code line  (-(P[i]). (P[i-1]). (P[i+1)))
                j:= j+1;  ō[j]= code line (m.j. + ∈ 2j̅. + ∈ 2j+1̅);
                P[i-1]:= L, j;  P[i]:= P[i+2];  go to Ex 7  end
                if logical (P[i]) ∧ Boolean (P[i-1]) ∧ Boolean (P[i+1])
                 then
        begin   n:= j+1
```

```
                q: = field (2, P[i-1])
                r: = field (2, P[i+1])
                t: = field (1, ō[r])
                field (1, ō[r]): = field (1, ō[q])
                if P[i] = 'Y'
                then
        begin   s: = field (4, ō[q])
                field (4, ō [q]): = t
        Ex 5:   if chain (s)
                then
        begin   u: = s
                s: = field (4, ō [u])
                field (4, ō [u]): = t;  go to  Ex 5  end
                field (3, ō [r]): = chain tag + q end;
                if P[i] = 'Λ'
                then
        begin   s: = field (3, ō[q])
                field (3, ō [q]): = t
        Ex 6:   if chain (s)
                then
        begin   u: = s;
                s: = field (3, o [u])
                field (3, ō [u]): = t;  go to Ex 6  end
                field (4, ō [r]): = chain tag + q end
                P [i-1]: = P [i+1]; P[i]: = P [i+2]; go to Ex7 end;
```

```
            if P [i] = ';'  P [i+2] = ','
            then
begin       ō[j]: = code line (t,2. +. t,2. t,1); j:=j+1
            ō [j]: = code line (t,2. +. t,2. e(1): j: = j+1
            ō [j]: = code line (t,2. abs. t,2.—)
            P [i]: = ','  go to Ex 7  end
            if P [i] = ';'   P[i+2] = ']'
            then
begin       ō [j]: = code line (t,4. +. t, 4. t, 3); j:=j+1
            ō [j]: = code line (t,4. +. t,4. e(1); j:=j+1
            ō [j]: = code line (t,4. abs. t,4.—);
            P [i]: = , ;  go_to Ex 7  end
            else
            if P [i] = ϒ then go to expression.
            if P [i] = ','
            then
begin       if procsw [k]
            then
begin       j: = j+1; ō [j]: = code line (I. Ω. (P[i-1]))
            j: = j+1; ō [j]: = code line (¬go_to Q[k]); P[i-1]:=L[k];
            j: = j+1; ō[j]: = code line (blank)
            L [k]: = j  end
            I [,*];  go to Ex 7  end
            if P[i] = ')'
            then
begin       if procsw [k]
            then
```

```
        begin  j: = j+1; Ō[j]: = code line (I.ℓ.(P[i-1]));
               j: = j+1; Ō[j]:= code line (—. L. | (R[k]));
               j: = j+1; Ō[j]:= code line (go to (L [k]).
               P[i-1]: = L[k]; k:=k-1; l: =i
Ex 4:   begin  i: = i-1; if P[i] = '('
               then
        begin  v: = i; m: = 1; go to Ex 5  end
               else
        begin  i: =i-1; go to Ex 4  end  end
Ex 5:   begin  j: = j+1; Ō [j] = code line (MT (P[i-1])
Ex 6:          if i = 1
               then
        begin  i: =ν+1; I [,c]; go to Ex 7  end
               else
        begin  j: = j+1; Ō[j] = code line (_st. (P[i+1]). ∏ (m))
               m: = m+1; i: =i+2; go to Ex 6  end
               P[i-2]: = P[i-1]; i: =i-2; I[,*], go to Ex 2  end  end
               if P[i] = ']'
               then
Ex 14:  begin  if P[i-2] =',' ∧ P[i-6] = 'dclar'
               then
        begin  j: = j+1, Ō[;]: = code line (t,2.*.t,2.t,4)
               j: = j+1, Ō[j]: = code line (t,1.*.t,1.t,4)
               j: = j+1, Ō[j]: = code line (t,1.t.t,1.t,3)
               j: = j+1, Ō[j]: = code line (t,1. +. Storage base. ≠,1)
               j: = j+1; Ō [j]: = code line (storage base, +.storage base,t,2)
               if proc nest ≠0
               then
        begin  j: = j+1, Ō [j]: = code line (e(P[i-5]), I,st. t,1)
               j: = j+1, Ō[j]:  = code line (e(P[i-5]),I+1. st.t,4)  end
               else
        begin  j: = j+1; Ō[j]: = code line (e(P[i-5]). st. t, 1)
               j: = j+1; Ō[j]: = code line ( e(P[i-5])+1.st. t,4 )
Ex 17:         if P[i-7] ≠ array then begin P[i-5]: =P[i-7]; go to Ex 14 end
               else go to expression end
               if P [i-2] = '[' ∧ P [i-4] = 'dclar'
               then
        begin  j: = j+1; Ō[j]: = code line (t,2.+.t,2.t,1)
               j: = j+1; Ō [j]: = code line (t,2.+.t,2. e (1))
               j: = j+1; Ō [j]: = code line (t,1.+.t,1. e (1))
               j: = j+1; Ō [j]: = code line (t,1.+. storage base. t,1)
               j: = j+1;Ō [j]: = code line (storage base. +. t,2.storage base)
               if procnest ≠ 0
               then
        begin  j: = j+1; Ō [j]: = code line ( e(P[i-3]), I. st. t,1) end
               else
        begin  j: = j+1; Ō [j]: = code line (e (P[i-3]). st. t,1) end  end
               if P [i-2] = ','
               then
        begin  if nondynamic (P[i-5]) ∧ procnest = 0
               then
Ex 12:  begin  j: = j+1; Ō [j]: = code line (t,1.*.t,1. e(P[i-5]) + 1)
```

```
                j: = j+1; $\overline{O}$ [j]: = code line ( I. +. t,1.t,2)
                j: = j+1; $\overline{O}$ [j]: = code line (t,1.st. e (P[i-5]), I ) end
                else
                if  dynamic (P[i-5])∧procnest = 0
                then  go to  Ex 12
                else
           begin  if nondynamic (P[i-5])∧procnest = 1
                then
           begin  if  local (P[i-5])
                then
           begin  j: = j+1; $\overline{O}$[j]: = code line (t,2.+.t,2. I)  end
                else
           begin  j: = j+1; $\overline{O}$ [j]: = code line (MT.BA)
Ex 13:          j: = j+1; O [j]: = code line (t,2.+.2.I)  end  end
                else
                if dynamic (P[i-5]) ∧ procnest = 1
                then
           begin  if  local (P[i-5])
                then
           begin  j: = j+1; $\overline{O}$[j]: = code line (t,1.*.t,1. e (P[i-5]+1). I )
                j: = j+1; $\overline{O}$ [j]: = code line (t,1.+.t,1.t,2)
                j: = j+1; $\overline{O}$ [j]: = code line (I. st. e(P[i-5], I )
                j: = j+1; $\overline{O}$ [j]: = code line (t,1.st.t,1,I)  end
                else
           begin  j: = j+1; $\overline{O}$ [j]: = code line (MT, BA); go to Ex 13 end  end
```

```
                  else
                  if P[i-2] = '['
                  then
           begin  if nondynamic (P[i-3] ∧ procnest = 0
                  then
Ex 14:     begin  j: = j+1; Ō [j]: = code line (I. st. t,1)
                  j: = j+1; Ō [j]: = code line (t,1.st. e(P[i-3]));
Ex 15:            P [i-3]: = t,1; i: = i+1, go to Ex 7  end
                  else
           begin  if dynamic (P[i-3]) ∧ procnest = 1
                  then
           begin  if P[i-3] = local
                  then
Ex 16:     begin  j: = j+1; Ō [j]: = code line (I. st. t,1,I)
                  j: = j+1; Ō [j]: = code line (t,1.st. e (P[i-3]))
                  go to Ex 15;  end
                  else
           begin  j: = j+1; Ō [j]: = code line (MT, BA);  go to Ex 16  end
                  else
           begin  if dynamic (P[i-3]) ∧ procnest = 0
                  then  go to  Ex 14
                  else
           begin  if dynamic (P[i-3]) ∧ procnest ≠ 0
                  then begin if local P[i-5]
                  then  go to  Ex 16
                  j: = j+1; Ō [j]: = code line (MT, BA);  go to Ex 1?  end
```

```
                else  go to alarm  end;  go to Ex 17  end
Ex 16:  begin   j: = j+1; $\overline{O}$ [j]: = code line (I. st. e P[i-3], I.␣)
                j: = j+1; O [j]: = code line (I. st. t, 1, I.␣ )
                go to  Ex 17  end
                if P[i] = 'then'
                then
        begin   q: = field (2, P[i-1])
                s: = field (3, $\overline{O}$ [q]);
                field (3, O [q]): = j+1;
Ex 9:           if chain (s)
                then
        begin   u: = s; s: = field (3, $\overline{O}$[u]); field (3, $\overline{O}$[u]): = j+1;
                go to  Ex 9  end;  I[,*],  go to  Ex 7  end
                else if P [i] = 'else'  then  begin q = field (2, P[i-3]);
                                              s: = field (4, O[q]);field (4,O[q]):
                                              = j+1;
                                              Ex 10; if chain (s) then begin u:=s;
                                              s:field(4, $\overline{O}$[u]);field(4,O[u]):=j+1;
                                              go to  Ex 10  end
                                              I[,*];  go to  Ex 7   end
                else if P[i] =; then  go to  statement
                else if P[i]= 'end'  then  go to statement
Ex 18:          if P[i] = '(' then  go to  Ex 19
                if P[i] = '[' then begin i:=i+1;  go to  EX 7  end
                go to  alarm
Ex 19:          if  P[i-1] = identifier
                then
```

```
begin k: = k+1; R[k]: = label
      j: = j+1; $\bar{O}$ [j]: = code line (go to  1 (R[k]))
      j: = j+1; $\bar{O}$ [j]: = code line (blank)

      L [k]: = j; procsw [k]: = true end

      I [,*]; go to Ex 7
```

Comment: Actual parameters of procedures are coded as follows:

```
      $j_\nu$:blank
      [Evaluation of
      parameter into t,1

      [address of t,1 into I

      go to $j_\nu$
```

These parameters will be entered from procedures via a MT command. The address j_ν, $\nu = 1,2,3,\ldots,q$ is stored in the currently available parameter position in the procedure parameter stack. Thus the total coding is

```
          go to  K
          code for parameter 1
            .....
          code for parameter q
          code for storing $j_1$ into $\Pi_1$
          code for storing $j_q$ into $\Pi_q$
      K: MT Procedure Name
```

7. The Analysis of statements

```
Compound : I [,*];     if I [,¢] ≠ letter then go to compound 1
                 A: = identifier accumulated
                 if I [,¢] = ':' then go to compound 3
                 go to compound 2
Compound 1: begin if I [,¢] = 'if' then go to if statement
                 if I [,¢] = 'go to' then go to go to statement
                 if I [,¢] = 'for' then go to for statement
                 go to alarm end
Compound 3: begin delta [12]: = 1; MT identifier declared;
                 O [,*]: = code line (␣. L. e (A))

                 go to compound end

Compound 2: begin If I[,¢] = '(' then go to procedure statement

                 If I [,¢] = ':=' then go to assignment statement

                 If I [,¢] = '|' then go to assembly code

                 If I [,¢] = '→' then go to assembly code

                 If I [,¢] = ';' then go to statement end

                 If I [,¢] = 'end' then go to compound end

                 If I [,¢] = 'blend' then go to block end end

Assignment statement: isrt ( O[,¢]); blocklist (O[,*]); O [,*]

                 go to expression analyzer
```

```
procedure statement: go to assign statement
if statement:  c:=1; isrt (2,O[c]); label (M[c]);
               next next (O[,c]: = code line (_. L.e(M[c]))
               list (O[,*]); isrt (7,O[,c]); label (M[c+])
               label (M[c+2]); list (O[,*]); isrt (2,O[,ϕ])
               o[,*]: = recognizer (Boolean);
               O[,*]: = eM[c+1]; O [,*]= e(M[c+2]);
               O [,*]: = code line (_. L. e(M[c+1]))
             O[,*]: = recognizer (unconditional statement)
               O[,*]: = code line (_. go to. e(M[c]))
               O [,*]: = recognizer (else)
               O [,*]: = code line (_. L. e(M[c+2]))
               O [,*]: = recognizer(statement)
               O[,*]: = code line (_. go to. e (M[c]));
               head (O[,c]; go to master control
```

Comment   The list structure built up for if statements has the form shown in snapshot form below. Z1, Z2, Z3 are label equivalents generated by label. >B<, >US<,>else<, and >S< are the inserted recognizers for Boolean, unconditional statement, else, and statements, respectively.

```
↑,  , L. Z1

, ( ,  ), L. Z1,
  ↑
,  ( , , , , , , , , ), L. Z1,
   ↑
, ( , ( , ) , , , , , , , ), L. Z1,
      ↑
 (, (, , , ), , , , , , , ), L. Z1,
    ↑
```

```
                   (, (, >B< , Z2, Z3), , , , , , , ), L. Z1,
                   (, (, >B<, Z2, Z3), L. Z2, >US<, go to Z1, >else<, L. Z3,
                   >S<, go to Z1), L. Z1, ;
for statement  begin:I [,*]; if I[,¢] ≠ letter then go to alarm
                    A: = identifier accumulated
                    X: = A; isrt (2,O[,¢]);
                    c: = c+1; d: = c; label (M[d]; c = c+1: e: =c
                    label (M[e]); next (2) (O[,¢]: = code line
                    (_. L. e (M[e]). _); list (O[,*]); isrt (3,O[,¢]) next (1)(O[,¢])
                    : = recogniser (forlist); next (2) (O[,¢]): = code line
                    (_. L. e (M[d]). _); next (3) (O[,¢]): = recogniser
                    (compound); block list (O[,*]); go to Ex 7 end
for list            if P[i] = ',' P[i-2] ≠ 'until' ∧ P[i-2]≠'while' then go to
                                                                     comma
                    if P[i] ='step' then go to step
                    if P[i] = 'until' then go to until
                    if P [i] = ',' ∧ P[i-2]='until' then go to until 1
                    if P[i] = 'while' then go to while
                    if P[i] = ',' ∧ P[i-2] = 'while' then go to until 1
                    if P[i] = 'do' ∧ P[i-2] = ',' then go to do 1
comma:         begin isrt (3,O [,¢]) :       O[,*]): = code line
                    (_. MT. e (M[d])); block list (O [,*]); next ()[c]):=
recogniser(forlist) ; isrt (2, I[,c]); next (2) (I[,¢]):= ':=';
```

```
                             I[,*]: = Z ; go to Ex 7 end
step            begin        isrt (5,O[,¢]); f: = c+1; label (M[f]); g: = c+1
                             label (M[g]); O[,*]: = code line (_. go to. e (M[f]))
                             O [,*]: = code line (_.L. e (M[g])); O[,*]: =
                             code line (_. MT. e (M[d])); isrt (4,I[,¢])
                             next (1) (I[,¢]): = X; next (2) (I[,¢]):= ':='
                             next (3) (I[,¢]): = X; next (4) (I[,¢]): = '+'
                             blocklist (O[,*]); next (O[,¢]) := recognizer (forlist)
                go to        Ex 7 end
      until     begin        isrt (6,O[,¢]);
                             O[,*]: = code line (m. st. t,l. ——)
                             O[,¢]: = code line ( __. L. e (M[+3])); variable (Q[p]).
                             isrt (4,I[,¢]); next (1) (I[,c]): = Q [p])
                             next (2) I[,¢]): = ':=' ; next (3) (I[,c]: = X
                             next (4) (I[,¢]): = '__'
                             next (2) (O[,¢]): = code line (t,l.*,m.Q [p])
                             next (3) (O[,¢]): = code line (—. ≤. t,l.0); c:=c+1
label (M[c]);                next (4) (O[,¢]): = code line (—.j. e(M[q]). e (M[c]))
                             next (5) (O[,¢]): = code line (—. L. e (M[c])
                             next (6) (O[,¢]): = recogniser ( forlist)
          block list         (O[,*]); go to Ex7 end
until 1;        begin        isrt (2,O[,¢]; next (2) (O[,¢]): = recogniser
          (forlist); block list (O[,*]); isrt (2,I[,¢]);
                             next (1) (I[,¢]): = X; next (2) (I[,¢]): = ':='
                            go to Ex 7 end
           while:begin isrt (6,O[,¢]);O[,*]: = code line (_. L. e (M[f]))
                             next (2) (O[,¢]): = code line (_. =. t,l. true); c:=c+1
```

```
label (M[c]);          next (3) (O[,¢]): = code line (_. j. e (M[¢]). e (M[¢]))
                       next (4) (O[,¢]): = code line (_. L. e (M[c]))
                       isrt (1,I[,¢]): = next (1) (I[,¢]): = 'if'
                       next (5) (O[,¢]): = recognizer (forlist); block list (O[,*])
                       go to Ex 7 end
          do 1: begin  isrt (2,O[,¢]); O[,*]: = code line (_. MT. e (M[d]))
          do 3:        O [,*]: = code line (_. go to. e (M[e])); go to O [,*] end
          do 2: begin  isrt (1, O[,c]); go to do 3
```

Comment The for statement is the most complex control statement in 20 ^ L.
Thus the statement for i:=E_1, E_2, step E_3 until E_4,E_5 step E_6 while B_7,E_8 do S
would generate code controlled as follows:

```
   i:= E1      generated by for
 -----------
   MT d
  i i=         generated by comma
 ---------------
       E2 <-+  generated by expression analyzer
 ------------|
   go to f   |
   label g   |  generated by step
   MT d      |
   i=i+      |
 ----------- |
        E3 <-|
   m:=t,1    |
   label f   |  generated by until
   p = i -   |
        E4 <-|
   if        |
   p * m ≤ 0 |
   then      |
   go to g   |
   i:=       |  generated by until 1
        E5 <-+
go to f'
label g'  generated by step
MT d
i = i +

    E_6

label f'
if

    B_7    generated by while

= . t,l. true
j. g. c'
label c'

i: =      generated by until l

    E_8

MT d
go to e   generated by do l

label f
statement
label e continuation
```

go to statement: P [1]: = 'go to'; go to expression analyzer

statement end: go to master control

compound end: go to master control

block end: begin

Comment The following code unravels all linkages between identifiers established in the current block. The example

$$\begin{array}{cccccccccccccccccccccccccccc} \underline{x} & \underline{y} & (& \underline{w} & x & w & x &) & x & w & \underline{z} & (& \underline{z} & y & (& \underline{x} & x & y & x &) & w & z & x &) & z & \underline{y} & \underline{w} &) \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \end{array},$$

where _ means declaration and (,) mean block beginning and ending, respectively and the order of occurrence being left to right would cause the following sequence of actions:

1. Assign x
2. Assign y
(3. Block begin
4. Assign w
5. Enter x
6. Already assigned w

```
    7. Already entered x
)   8. Block end, so: chain unassigned x to assigned x (1.)
          remove block begin (3.)

    9. Already assigned x  through chain
   10. Enter w
   11. Enter z
(  12. Block begin
   13. Assign z
   14. Enter y
(  15. Block begin
   16. Assign x
   17. Already assigned x
   18. Enter y
   19. Already assigned x
   20. Block end. so: chain unassigned y to preceeding unassigned y
       remove block begin (15.)
   21. Enter w
   22. Already assigned z
   23. Enter x
)  24. Block end.  So chain unassigned x to assigned x through chain (1)
                   chain unassigned w to unassigned w (10.)
                   chain unassigned y to assigned y (14.)
                   remove block begin (12.)
   25. Enter z and chain (11.)
   26. Assign z through chain (25.)
   27. Assign w through chain (21)
)  28. Block end.  So all assigned  Remove Block begin;

Block end:   begin   H: = K-1; if unassigned ([illegible]) then go to Bl 1

             Bl 2:  if marker       ) ≠ '('

                    then begin      H-1; go to Bl 2  end

                    if number       > length [s]

                    then length [s]: = number [r]

                    go to Block end 1

             Bl 1:  Mu: = H

             Bl 3:  1  if chain (T[MU]

                    then begin MU: = field (5, T[MU]); go to Bl 3  end
```

```
                Bl 4:    if field (3, T[T]= field (3,T[MU])
                         then  go to Bl 5
                Bl 6:    if field (2, T[T]) ≠ '('
           then Bl 7:    begin T: = T-1;  go to  Bl 4  end
                         Q: = Q-1
                         if Q = 0  then  go to  Bl 2
                         go to Bl 7
                Bl 5:    if unassigned (T[[T]
                         then begin field (5,T[MU]) : = T;  go to Bl 2  end
                         if block (T[T]) ≥ crtblk
                         then  go to  Bl 6
                Bl 8:    MU: = H
                         field (4, T [MU]): = field (4, T[T])
                         number [r]: = number [r] + norm field (3, T[MU]))
                         if  field (4, T[MU]) = 0
                         then begin  MU: = field (4,T[MU]);  go to Bl 8 end
                         field (4, T[MU]): = T;  go to  Bl 2  end
Block end 1:             if  S: = 0
                         then  go to Pass 1 end
                         S: = S-1;  go to  master control
Master control:          0 [,*]; if recognizer (O[,c])
                         then  go to recogniser
                         go to  block
```

Comment The table T is the identifier table. Its field structure is <line number>. <block marker>. <identifier>. <block number>. <chain number>. <indirect>. <delta vector>.

```
identifier declared:   ; if field (2, T[K]) = true
                         then go to ID 2
                         H: = K
          ID 3:          H: = H-1; If field (3, T[H]): = A
                         then go to ID 4
          ID 5:          if field (2, T[H]); = true
                         then go to ID 2
                         go to ID 3
          ID 2:          field (1, T[K]: = K; field (3, T[K]]): = A
                         field (4, T[K]): = crtblk; field (7, T[K]]): = delta
                         K: = K+1; go to identifier declared
          ID 4:          if field (4, T[H]) = 0
                         then go to ID 7
                         MU: = H
          ID 6:          field (4, T[MU]: = crtblk
                         if field (5, T[MU]) = 0

                         then go to identifier declared

                         MU: = field (5, T[MU])
                         go to ID 6
          ID 7:          if param = 1
                         then begin field (7, T[H]): = delta; go to identifier de-
                                                               clared end
                         go to ID 5
Identifier encountered but not declared:    ;
```

Comment This routine is entered automatically whenever any identifier is encountered other than in a declaration. The identifier is A ;

```
          H: = K-1
```

```
        IE 1:        if field (3, T[H]) ≠ A
                     then go to IE 2
                     if field (4,T[H]) = crt blk
        I[ 3:        then begin correspondant: = H;IE 4: go to identifier

                     encountered but not declared end
                     if field (4, T[H]) = 0
                     then go to IE 3
        IE 2:        if field (2, T[H]) ≠ '('
                     then begin H:=H-1; go to IE 1 end
                     field (3,T[K]): =A; field (7,T[K]): =
                     union (field(7,T[K]), delta);K:=K+1; go to IE 4

    Chain :   ;      H: = K-1

    Chain 1:         if field (3,T[H]) = field (3,T[K])

                     then begin

                     field (4, T[K]) = field (4, T[H])

                     field (5,T[K]) = H; go to chain end

                     H:= H-1
                     if H=0 then go to alarm
                     go to chain 1
```

Assembly code:

begin comment This code is machine code and its syntax has been described in Part I. Basically the format is operator | operand or operator → operand;

```
                  if machine operand (A)
                  then  go to  AC 1
                  if macro (A)
                  then  go to AC 2
                  go to alarm  end
AC 2:             list (I[,ȼ]);
                  for Z: = 1 step 1 while MT (Z] ≠ terminal signal do
begin             if field (1, MT [Z]) = A
                  then begin H: = field (2, MI[Z]);  go to AC 3  end  end
                  go to alarm
AC 5:             seqe (V, AC 6);
AC 10:            V[,*]; A: = V [✓,c]; seql (G, AC 10);
AC 9:             G[,*]; def (W,G[,ȼ]);  if A=W[✓,*] then go to AC 8;
                 else  go to  AC 9;
AC 9:             copy (W[ ,*], V[,c]);  go to  AC 10
AC 6:             copy (V,I[,*]);  go to statement
AC 3:             Copy (MT[n], V); list G; seq W (G, AC 5); seqw w(V,AC 5)
                  for Z:=1 step 1  while  MI[Z]≠0 do
                  begin isrt (,G[,ȼ]); G[✓,*]: = MI [Z]; I[,*]; S:=1
AC 11             list (G[,ȼ]); if I[,*]='('
                  then begin S:=S+1; AC 12: isrt (G[,ȼ]); G[,*]:=I[,*] end
if I [,ȼ] = ')' then begin S:=S-1;  go to AC 12 end
                  go to AC 12
AC 12             if S:=0  then  go to AC 11
                  go to AC 12
AC 11:  ;         end
```

AC 1: begin comment This code section analyzers machine assembly code. It uses the address expression analyzer. The constituents of address expressions are identifiers, integers, the arithmetic characters + and -. The address expressions are machine dependent and the analysis given is for the Bendix G-20.

```
AC 25        isrt (O[,¢])
AC 35        if I[,¢] $\mathcal{F}$ simple variable+OA ∨ I[,¢]$\mathcal{F}$OA + simple variable
             then go to AC 26
             if I [,¢]$\mathcal{F}$(simple variable + OA)
             then  go to  AC 27
             if I[,¢]$\mathcal{F}$ (OA + simple variable)
             then go to AC 27
             if I[,¢]$\mathcal{F}$ (simple variable + OA)
             then go to AC 30
             if I[,¢]$\mathcal{F}$ (OA + (simple variable))
             then  go to  AC 30
             if I[,¢]$\mathcal{F}$ (simple variable) + OA
             then  go to AC 32
             if I[,¢] $\mathcal{F}$ OA + (simple variable)
             then  to
             then  go to AC 32

             if I[,¢]$\mathcal{F}$simple variable + simple variable

             then  go to AC 34

             go to  alarm
```

Note: similar code need be written for occurrence of (_).

```
AC 26:       MT  AC 36
             O[,*]: = code line (_. OCA(O)., e (A)., _)
AC 26'       I[,c]: = 'OA';  go to AC 25
AC27:        MT  AC 36
```

```
AC 30:    MT AC 36
          O[,*]: = code line (_. OCA(3). G(A). _);  go to AC 28
AC 32:    MT AC 36
          O [,*]: = code line (_. OCA (1). e (A)._); go to AC 28
AC 34:    MT AC 36
          O [,*]: = code line (_. OCA(0). e (A)); go to AC 28
AC 36:    ;A: = simple variable identifier accumulated;
          go to AC 36
```

Appendix 1.

1.1 In declaration s add

```
Switch 1:  If I [,∉] = letter then go to switch 5
           go to alarm
Switch 5:  A: = identifier accumulated; delta [12]: = 1
           MT identifier declared; SW:=A; mu := 0
           P [1]: = 'switch', go to expression analyzer
```

1.2 In the expression analyzer add

1.2.1 code relating to switch

```
          if P[i] = ':='  P[i-2]: = 'switch' then go to switch 3
          go to Ex 7
Switch 3. P [i] : = [i-2]; i: - i+1; go to Ex 7
          if P[i] = ','   P[i-2] = 'switch'  then  go to switch 4
Switch 4  begin if simple (P[i-1])
          then begin code constant ( e(sw) + mu: = code line (_. go to.
           (P[i-1]). _))  mu: = mu +1 end
           else begin code constant (e(SW) + mu: = code line(_.go to.e(TSW).
                                                                          _))
           Ō[j]: = (_. go to.0,I._)  end
           go to  switch 3 end
           if P[i]: = 'switch' then
           begin  label (TSW); isrt (2,C[1,c])
           Ō [,*]: = (_.1.e(TSW))
           block list (O[,*]) end
           go to Ex 7
```

1.2.2. code relating to go to statements

```
             if (P[i] = 'T' ∨ P[i]=';' ∨ P[i]= 'end') ∧ P[i-2] = 'go to'
                     begin
             then if local (P[i-1]) then  go to  go to 1
                                         else  go to  go to 2 end
             go to  Ex 7
go to 1:     begin j: = j+1;  Ō [j]: = code line (_. go to. e (P[i-1])._)
             go to  statement end  end
go to 2:     begin j:= j+1; )[j]: = code line (I. ℓ. e(P[i-2])._)
             ETA: = block (P[i-1])
             j: = j+1; Ō [j]: = code line (I. ℓ. e (ETA)._)
             j: = j+1; Ō [j]: = code line (_.go to. BA go to._)
             go to statement end  end
```

PART III

A Programming Manual for $20^{\wedge}L$

1. Introduction

$20^{\wedge}L$ is a programming language for -- in the main -- scientific computation. There is, in the language, no extensive input/output facilities. These are provided by procedures but a few sample such will be mentioned in the sequel.

$20^{\wedge}L$ is one language with one processor even though the progeammer may write in 3 different modes: algebraic language, assembly language, or list language and they may be mixed as desired by the programmer. Thus, for programs requiring -- for reasons of speed and efficiency of storage (word packing) -- total control over the machine's abilities the programmer may, in continuous transition, skip into symbolic machine language.

On the other hand, if he is doing extensive symbol manipulation he may choose to program more extensively in a list formalism.

Most importantly as a program is debugged it can be altered from the language which it is easiest to test the logic of the program into that in which it is most efficient to operate the program.

2. Programming Principles

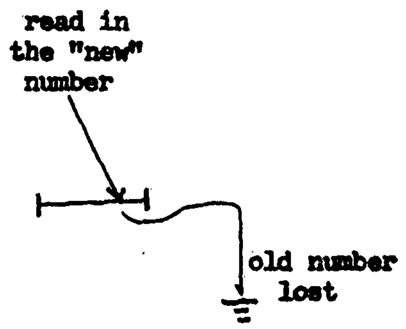
The languages used on computers follow very closely two fundamental principles of computer design:

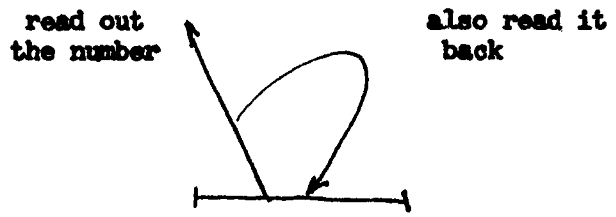
(i) The nature of storage

and (ii) The sequencing of control

The computer's storage is divided into units called "words." These are of fixed length--or sometime small multiples of units of fixed length. Each word can be identified by a natural number called an address which, by the nature of mechanical devices has a range, e.g., from 0 to 2^{15}. Words store numbers and consequently we may say that an address is the name of a problem variable if the contents of the word having that address change in the manner specified by operations on that variable.

The storage is further characerized by destructive read-in and non- destructive read-out meaning that each time in formation is stored in a word the previous contents are replaced by the new information. However, when information is read from the word the contents are restored on read-out. Thus:





In some applications the programmer constructs, through programming, a pseudo-memory called a stack or push down list. In this memory each time a word is read into the same "cell" the previous contents are not lost but are pushed down deeper into the stack. Here, reading can be destructive or not as wished. But this is accomplished by programming and not by hardware. It is particularly easy to accomplish using Tlists,

The second characteristic is sequencing of control.

Each instruction has an identifier called its label or name (absence implies a blank label) and an operation part.

In the computer each time an operation is completed a next is chosen according to the rule:

(i) if the operation does not specifically identify the name of its successor the lexicographic next is next to be executed.

(ii) if it does identify its successor then that one so identified is next executed.

The source of computer flexibility is that the choice of (i) or (ii) in any instance may be made conditional on the consequences of already accomplished computation.

In the case of 20^ the sequencing rules are slightly more complicated by the concept of the control statement. The control statement inducesa a sequencing control over the statements within its scope. The scope is defined as the set of all statements following until a punctuation convention is satisfied. The lexicographic last is called the terminal statement. Then the above sequencing rules hold with the additional rule:

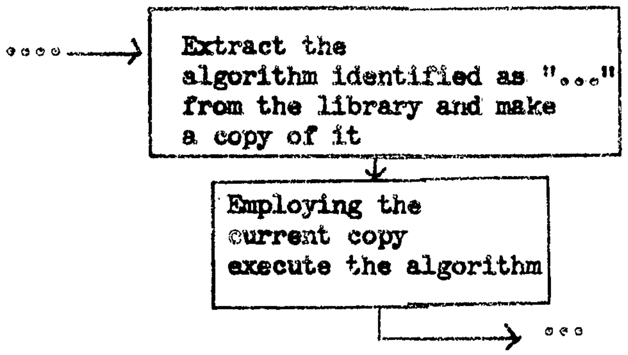
(iii) if the current statement is the terminal of a control statement, the successor is determined by the control statement.

Thus in 20^L programming the programmer must be constantly aware of this inter play between sequencing and assignments of values to variables by which his computational purpose is advanced.

There are two fundamental aspects to programming in 20^: one is the programming of cycles or loops and the other is the analysis of arithmetic expressions. The semantics of the language has already been discussed and this manual is thus concerned with principle and example.

1. Programming of cycles or loops.

Almost without exception every algorithm executed on computers contains at least one cycle. For, without a cycle, every step of the algorithm would be executable at most one time. The execution time of the algorithm could then be of the same order as the time required to describe it. Even for algorithms without explicit cycles, its availability in a standard form in a library (from which reference and extraction are possible) imbeds the algorithm in the "library cycle":


....
Extract the algorithm identified as "..." from the library and make a copy of it
Employing the current copy execute the algorithm
...

For each use the re-description time is effectively zero. We turn our attention to cycles within algorithms.

A chain (of instructions) is a sequence of instructions $I_1, I_2, \ldots, I_n$ such that for each k, $2 \le k < n$, I_k is the successor of I_{k-1} and the predecessor of I_{k+1}.

A chain which starts and ends at the same instruction is called a cycle.

A cycle of instructions clearly permits the same sequence to be carried out several times. In programming we note the obvious constrants on cycles so that they are not carried out an infinite number of times:

(1) Every cycle must possess a branch or comparison instruction (else it could never terminate), and

(2) At least one storage location must change its contents during the course of a cycle satisfying (1).

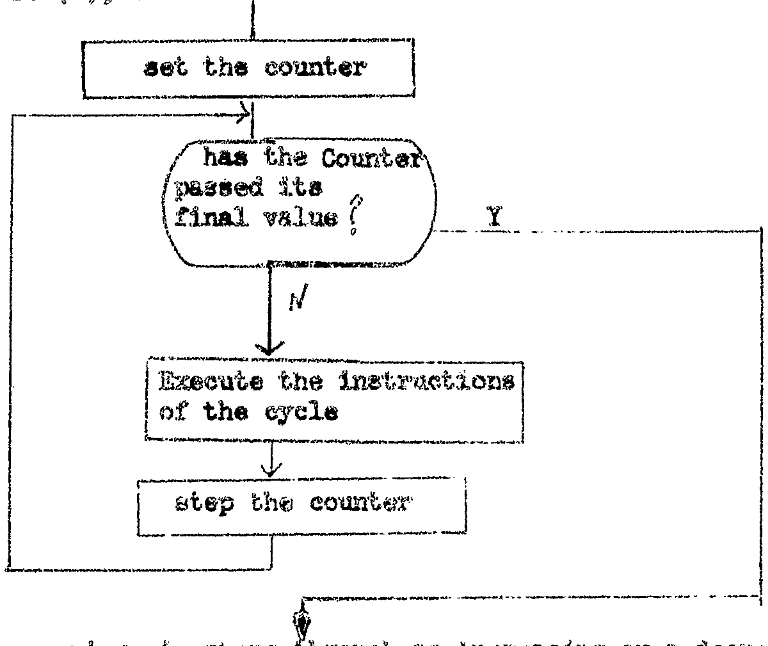
The termination condition is of great importance and is one of, or combination

[illegible]

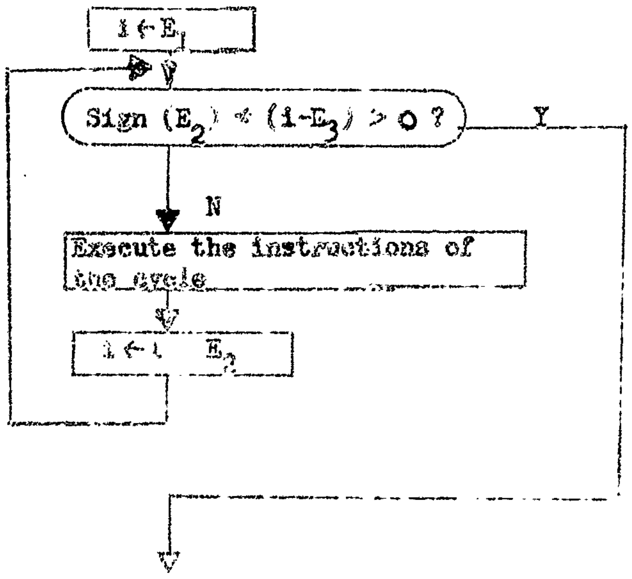
(a) A counter achieves its final value after stepping through a sequence of intermediate values.

(b) A relation is first satisfied (or not satisfied), e.g., $x < y$, or $(x \neq y)$ and $(z \leq y + 3)$, etc.

In the case (a), the flow chart is of the form:


set the counter
has the Counter passed its final value?
Y
N
Execute the instructions of the cycle
step the counter

In general the counter, i, steps through an increasing or a decreasing sequence of values with the initialization, the progression, and the termination determined by fixed functions. An obvious notation is: **for** $i = E_1$ **step** E_2 **until** E_3 **do** < instructions of the cycle> The chart becomes:



Sometimes the cycle continues while an arithmetic relation remains satisfied. In these cases we may write:

++ <u>for</u> i ← E_1 <u>step</u> E_2 <u>while</u> R

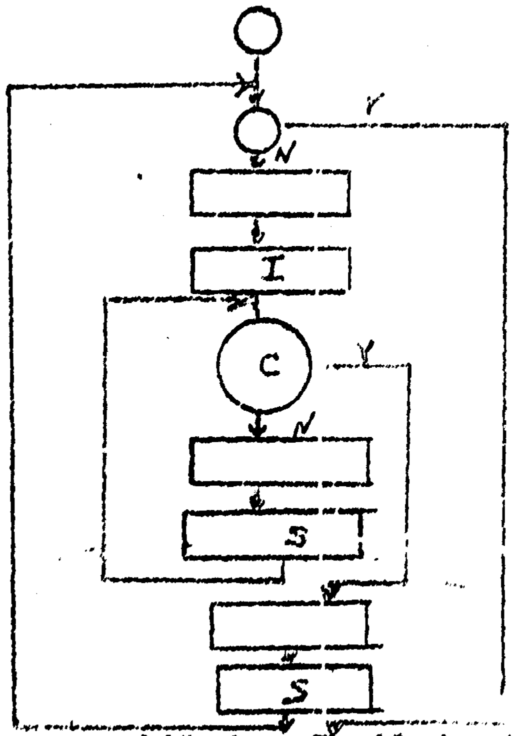
Exercises:

1. Draw the cycle chart for case (b) of page 101.1

2. Draw the cycle chart for the description ++.

3. How would the description ++ be modified for the case of cycling until R first becomes satisfied?

In many cases cycles are nested, i.e., within the instructions of the cycle is the complete specification of another cycle. Using outline notation:



represents two nested blocks. The blocks which must be present are labeled I: initialize; C: check; S: step. Using parentheses to indicate cycle relationships we find situations like: (()) as above, but also (([illegible] ([illegible] and combinations [illegible] but we <u>do not use</u> overlapping cycles such as [illegible]

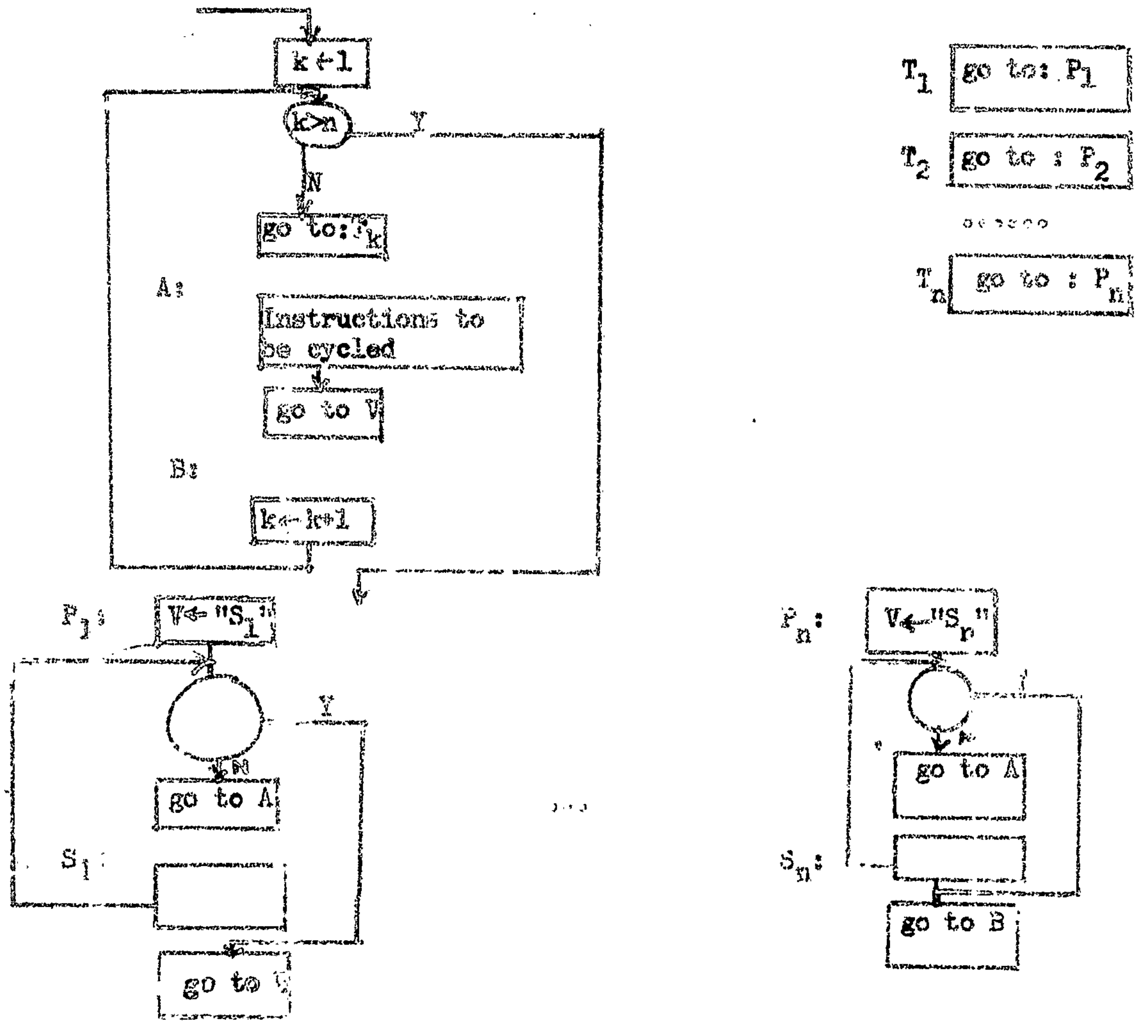
Exercises:

1. Suppose we use [illegible] to represent the beginning and end of a cycle called [illegible] [illegible]

any sequence of such symbols to determine if it is an allowable nested sequence.

2. Modify 1. so that the maximum depth of nesting in such sequences is computed, e.g., (((())((())))) has a maximum depth of 5.

Often several progressions, say n, share the same set of instructions over which to cycle. This case may be treated in a simple manner by inducing a <u>master</u> cycle, k, which progresses in steps of 1 from 1 to n selecting and applying each of the progressions in turn to the instructions overwhich to cycle. The chart involves a selection switch, T, having n steps:

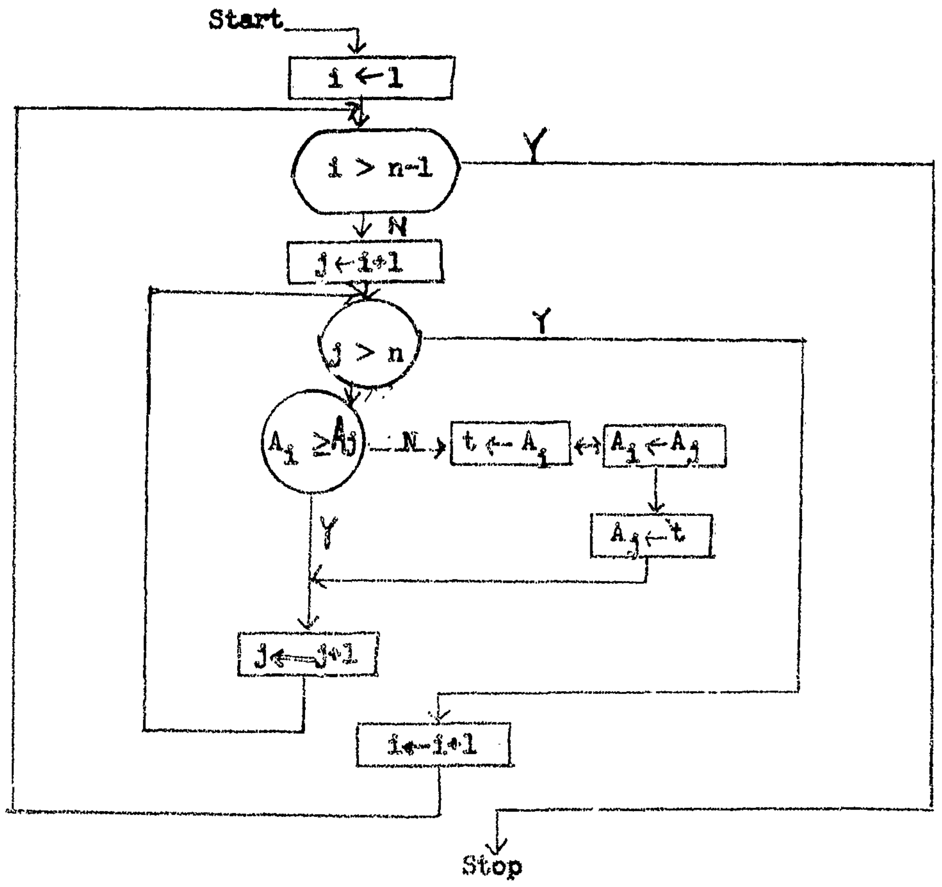


Progressions with gaps can be treated by this technique.

Examples of cycling:

1. Nested cycling

Order the numbers (in) $A_1, A_2, \ldots, A_n$ so that $i > j$ implies that $(A_i) \geq (A_j)$.



2. Shared cycling

Outside of progressions with gaps it is difficult to select a simple example using shared cycling. Nevertheless, consider the problem to compute:

f ([illegible]) for $0 \leq X \leq 5$ in steps of 0.1, i.e.,

for $X = 0\ (0.1)\ 5$, where

$$f(z) = \begin{cases} z, & 0 \le z \le 1 \\ \frac{1}{z} + (z-1)(z^2 + \sqrt{z}), & 1 \le z \le 2 \\ (\frac{1}{z} + \sqrt{z}) + (z^2 + 2(z-2)), & 2 \le z^2 \le 10 \end{cases}$$

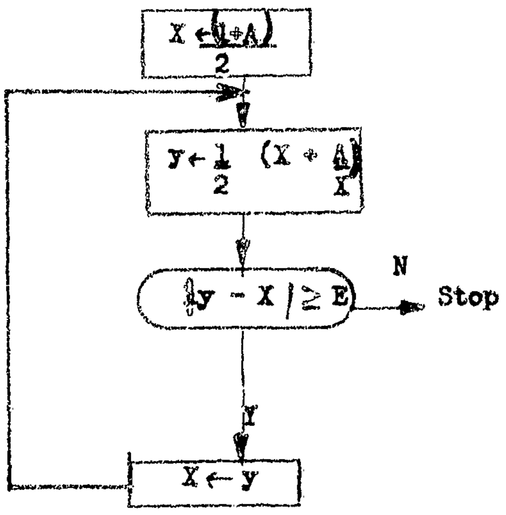
Otherwise undefined

and

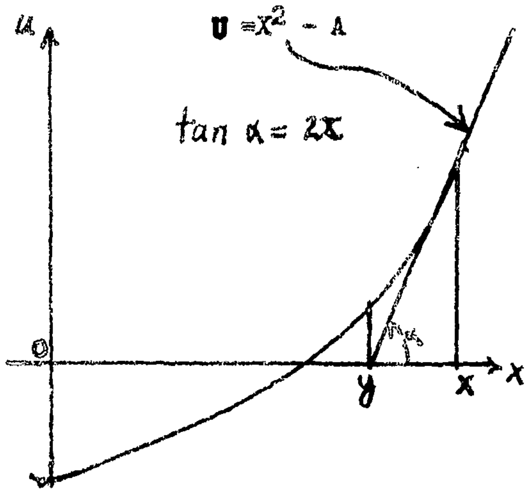
$$z = X^2 + \sum_{\nu=[0/X]}^{1000} \frac{1}{\nu^2}, \quad \nu \text{ integral.}$$

Some of the cycling here can be shared.

The case where the extent of cycling is determined by a relation is treated in a similar way, except that a cycle counter may or may not be present. Thus:



$X \leftarrow \frac{(1+A)}{2}$
$y \leftarrow \frac{1}{2}\left(X + \frac{A}{X}\right)$
$|y - X| \ge E$
N
Stop
Y
$X \leftarrow y$



cycles, not on a counter, but <u>while</u>(meaning as long as) $|y - X| \ge E$. The above chart represents Newton's method for finding the square of A.

Exercise:

1. Add a counter to the above chart so as to count the number of cycles required before exiting.

2. For all integers 1, 2, ..., N generate the sequence of integers I_i which are prime, i.e., possess only 1 and themselves as factors. Hint: If X is an integer its largest factor different from itself must be $\leq \sqrt{X}$.

3. Suppose a continuous function f (X) is specified for $a \leq X \leq b$ and that $f(a) * f(b) < 0$. Let the X- intercept of the straight line between (a, f (a)) and (b, f (b)) be a first approximant of a zero of f(X). Using a sequence of such straight lines develop a sequence of approximants terminating the process as soon as two successive elements of the sequence differ in magnitude by less than a given $\delta > 0$.

2. Propositions or relations.

An influential component of every computation is the set of discriminations which influence the branching and eventually determine terminating conditions. Each such discrimination is a question which is, each time, answered yes or no. But each such question is also a proposition having a value true (T) or false (F). Since propositions take on only two values these may be represented by 1 (T) or 0 (F). Thus the question: "Is X> 7? may be treated as a proposition P (X) = (X>7) which in reality is a propositional function.

If E_1 and E_2 are arithmetic expressions, e.g., X + 9 * Z and $n*\sqrt{n-s*(9+y)}$, and R is an arithmetic relation, e.g., $>, <, \geq, \leq, =, \neq$, then ($E_1$ R E_2) is called an elementary (arithmetic) proposition.

Since propositions take on the values 0 and 1 it is convenient to define propositional variables which take on only these values. This permits us to write equations like:

$$b \Leftarrow (E_1 \ R \ E_2)$$

Propositional variables are sometimes called Boolean variables; they, too, are elementary propositions.

Compound propositions are formed from elementary propositions by combining the latter under basic propositional operations. These basic operations are:

(1.) Complementation ($\neg$): If P is a proposition then so is $\neg$P defined as:

$P = 0 \longrightarrow \neg P = 1$, $P = 1 \longrightarrow \neg P = 0$, or in tabular form:

P	$\neg$P
0	1
1	0 .

$\neg$ satisfies: $\neg(\neg P) = P$.

If P and Q are propositions then so are:

(2) P$\vee$Q defined by the table:

P	Q	P$\vee$Q
0	0	0
0	1	1
1	0	1
1	1	1

$\vee$ is called the <u>inclusive</u> or, i.e., P$\vee$Q is true , if either or both of P and Q are true.

(3). P$\wedge$Q defined by the table:

P	Q	P$\wedge$Q
0	0	0
0	1	0
1	0	0
1	1	1

$\wedge$ is called <u>and</u>, i.e., P$\wedge$Q is true if and only if P is true and Q is true.

(4) P $\supset$ Q defined by the table

P	Q	P $\supset$ Q
0	0	1
0	1	1
1	0	0
1	1	1

$\supset$ is called implication.

(5) $P \equiv Q$ defined by the table

P	Q	$P \equiv Q$
0	0	1
0	1	0
1	0	0
1	1	1

$\equiv$ is called equivalence i.e., $P \equiv Q$ is true if and only if P and Q have the same value.

The above tables are called truth tables and each binary propositional operation defines and is defined by such a table.

Unlike binary functions, involving operations on real numbers to form real numbers of which there are non-denumerably many, there are only a finite number of binary operations mapping propositional variables to propositional variables.

Exercises:

1. How many such binary propositional operations are there?

2. If P, Q, R, are propositional variables how many propositional functions of 3 variables are there?

Hint: Start the counting analysis from the truth table.

3. If $\Diamond$ is used for exclusive or i.e., either (but not both of) P and Q are true for $P \Diamond Q$ to be true, construct the truth table and represent $\Diamond$ in terms of (1), (2), (3).

It is often quite convenient to synthesise propositional expressions from the truth table. In order to do this we investigate some properties of these operations, particularly (1), (2), and (3). The following identities are easily proved by truth tables:

(1) $P \vee (\neg P) = 1$	(4) $P \wedge (\neg P) = 0$
(2) $P \vee 1 = 1$	(5) $P \wedge 1 = P$
(3) $P \vee 0 = P$	(6) $P \wedge 0 = 0$

Propositions obey the <u>distrubutive law</u> of $\wedge$ over $\vee$:

$$P \wedge (Q \vee R) = P \wedge Q \vee P \wedge R.$$

Both $\wedge$ and $\vee$ obey the associative law:

$$P \circ Q \circ R = (P \circ Q) \circ R = P \circ (Q \circ R);$$

and the commutative law

$P \circ Q = Q \circ P$, where $\circ$ means $\vee, \wedge$.

Consequently, e.g.,

(7) $P \vee (P \wedge Q) = P.$

By constructing the truth tables it is demonstrated that

(8) $\neg (P \vee Q) = \neg P \wedge \neg Q$ and

(9) $\neg (P \wedge Q) = \neg P \vee \neg Q.$

One defines that $\vee$ is the dual of $\wedge$ and vice versa; and $\neg\neg$ is the dual of $\neg$ and vice versa.

If S is a propositional expression involving only $\vee$, $\wedge$, $\neg$, and elementary propositions, then the dual of S is obtained by replacing, in turn, from left to right each occurrence of $\vee$, $\wedge$, and $\neg$ by its dual.

Example:

$$\begin{aligned} S &= (\neg P) \wedge (\neg(Q \vee \neg S)) \\ &= (\neg P) \wedge (\neg Q \wedge S) \end{aligned}$$ Note: Q means $\neg \neg Q$.

then dual $(S) = P \vee (Q \vee \neg S) . = \neg S.$

A general theorem is that: $\neg S$ = dual (S) for any propositional function.

In order not to have to write parentheses to excess, as in the case of arithmetic operators, an assumed heirarchy of propositional operations is defined:

$\neg$ before $\wedge$ before $\vee$ before $\supset$ before $\equiv$.

Now consider the case of a propositional function, F, of, say, 3 variables P, Q, and R, i.e., F (P, Q, R) defined by the table of $2^3=8$ entries:

P	Q	R	F (P, Q, R)
0	0	0	f_0
0	0	1	f_1
0	1	0	f_2
0	1	1	
1	0	0	...
1	0	1	
1	1	0	
1	1	1	f_7

Where each f_i is 0 or 1. Then F can be "expanded" into a "sum" of eight "products":

$$\neg P \wedge \neg Q \wedge \neg R \wedge f_0 \vee \neg P \wedge \neg Q \wedge R \wedge f_1 \vee \neg P \wedge Q \wedge \neg R \wedge f_2 \vee \dots \vee P \wedge Q \wedge R \wedge f_7;$$

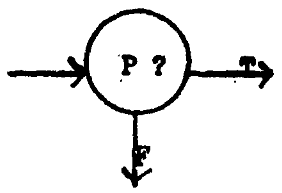
for any values given to P, Q, and R cause one and only one of the eight products to differ from zero (i.e., be 1) so that its value is $1 \wedge f_k$ and f_k is the function value corresponding to the given triplet of values for P, Q, and R.

Example:

P	Q	R	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

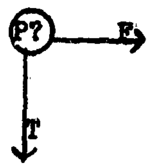
$$\begin{aligned} F &= \neg P \wedge \neg Q \wedge \neg R \vee \neg P \wedge \neg Q \wedge R \\ &\quad \vee P \wedge Q \wedge \neg R \vee P \wedge Q \wedge R \\ &= \neg P \wedge \neg Q \wedge (\neg R \vee R) \vee \\ &\qquad P \wedge Q \wedge (\neg R \vee R) \\ &= \neg P \wedge \neg Q \vee P \wedge Q \\ &= \neg (P \vee Q) \vee P \wedge Q \\ &= P \equiv Q \end{aligned}$$

In programming, propositional functions are used to determine branching patterns in programming. If P is a proposition we represent it in flow charts as:

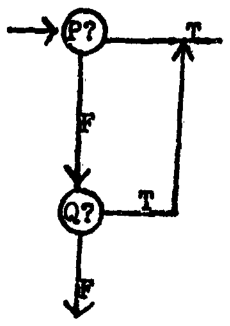


; and a reversal of labels is

P :

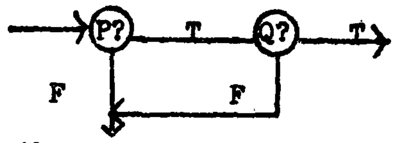


Then, PVQ is represented as



or that obtained in permuting Q and P. $P \wedge Q$

is



while $Q_\wedge P$ is that obtained in permuting Q and P.

While the propositional affect of PVQ and QVP is the same, we will see later there is sometimes reason to chooseone over the other.

We will now apply the foregoing to a specific example: Evaluate and print F (X, Y) for Y=0(1) 20 and for X=0(1) 20

In English: $F(X,Y)= F_1 (X,Y)= X^2+ [X^2/3]$ if $0 \leq X < 3$ or $7 \leq X < 11$ but

not X divides Y or X even; should it not be $X^2 + [X^2/3]$ then

$F(X,Y) = F_2(X,Y)= X^2 - X\cdot[\frac{X^2}{3}]/Y+1)$ unless $\frac{X}{2}$ is even in which case

$F(X,Y) = F_1 (Y,X) - F_2 (Y + 1,X)$.

Comments: (1) The above is somewhat ambiguous. It can be clarified by requiring its statement in an unambiguous problem language. (2) What precisely is meant by "for Y = 0(1) 20 and X = 0(1) 20"? Does it mean simultaneous or iterated cycling• What happens if $11 \leq X$? Is it the sense of the problem that the scope of the first or is "$\leq$ X 11", or "$7 \leq$ X 11 but not X divides Y", or the preceding coupled to "or X even"?

These and similar questions can be answered by the use of operators and parantheses, thusly:

```
            for  Y = 0 step 1 until 20 do
     begin  for  X = 0 step 1 until 20 do
     begin  if ( 0 ≤ X < 3  V 7 ≤ X < 11)  (divides (X,Y)V Divides (2,Y) )
            then  F := X↑2 + entier ( X↑2 / 3 )
     else begin  if    (divides(2,X/2)
            then  F := X↑2 - X x entier (X↑2/3) / (Y+1)
            else  F := Y↑2 + entier (Y↑2/3) - ( (Y+1)↑2 - (Y+1)x entier
                                                   ( (Y+1)↑2/3) / (X+1) )

       print ( F )  end end end
  halt;
```

4. **Encoding of algebraic formulae.**

Everyone is familiar with formulae. Some examples are:

a) $t = x + 4 * y$

b) $t = (x-z) * z + y/(2 + r))$

c) $z = c * (r + a/x)$

The expressions on the right are said to define the variables on the left. In computation the counterpart is that the right hand sides are evaluated, using the current values of the variables appearing there, thus defining a value for the variable on the left. This time sequence is emphasized by using a sign like $\leftarrow$ or := in place of = thus:

a) $t \leftarrow x + 4 * y$

or b) $t := (x - z) * (z + y/(2 + r))$.

with this sense of := (which will be used hereafter): a formula:

$t := t + x * t$

has computational meaning, i.e., defines a specified set of actions to be carried out. Thus, in Newton's method for computing the square root of "a", the formula

$$x := \frac{1}{2}\left(x + \frac{a}{x}\right)$$

yields the iterants successively, with each new one computed from the preceding one.

In computation each variable may be said to be the name of a storage location and its contents the (current) value of the variable. In the above, clearly, the new value of x reads over the previous value and unless else-w here retained it would be lost when read over.

Much of what we have learned about formulae is based on the assumption that formulae are generally simple, i.e., not too many symbols and not too many expressions involving repeated division or expoentiation. In computation these assumptions are no longer generally true. Thus while "x^2""is quite unambiguous, "$x^{2^{2}+1}$" is less clear unless the printing is very precise, and then what of $x^{2^{2^{2}+1}+1}$? Consequently, the formulae dealt with have the property that:

<u>All formulae are represented as finite linear sequences of characters.</u>

Thus such formulae do not contain either superscripts, subscripts, or exponents. Naturally a representation needs to be introduced to take their place. All such representations are based on a partition of the alphabet of reconizable characters into disjoint classes. For example, such a partition might be:

a) The set of operators: "+", "-", "*", "/", "↑", " :=".

b) The set of two sided delimiters: "(", ")".

c) The set of numbers , e.g., 2, 10.5, 110,.

d) The set of variables: X, MU, T[illegible]U3, etc.

These latter two classes m[illegible] described by example. How are they to be described explicitly?

In a subsequent set of notes a method for their formal description will be given.

Then a formula

$$Y := C * (A + D\ I + E^{2 + R^2})$$

could be represented by the linear character string:

$$Y := C * (A + D\ I + E \uparrow (2 + R \uparrow 2)) \tag{1}$$

where $\uparrow$ denotes exponentiation

The operators are binary; They have two associated operands, e.g., in (1) for the underlined operators the associated operands are:

$$A \underline{+} D\ I \tag{2.1}$$

$$R \underline{\uparrow} 2 \tag{2.2}$$

$$2 \underline{+} R \uparrow 2 \tag{2.3}$$

$$E \uparrow (2 + R \uparrow 2) \tag{2.4}$$

$$A + DI \underline{+} E\ (2 + R\ 2) \tag{2.5}$$

$$C \underline{*} (A + D\ I + E \uparrow (2 + R \uparrow 2) \tag{2.6}$$

and, even

$$Y \underline{:=} C * (A + DI + E \uparrow (2 + R \uparrow 2)) \tag{2.7}$$

The assignment of operands to operators is determined by:

(i) The direction of scan of the formula, e.g., from left to right;

and (ii) The heirarchy of the operators, e.g.,

$\uparrow$ before $*$ before $/$ before $-$ before $+$ before $:=$.

However all operations within a matching set of parentheses are accomplished before the operation for which the matching set of parentheses delimit an operand, as (2.3) preceeds (2.4).

Exercises:

1. **For the following, represent the formulae by linear strings of characters:**

$$A + \frac{}{C + \frac{D}{E * F * (R - M)}}$$

$$X := 3 \qquad\qquad\qquad\qquad + \frac{R}{C * L}$$

$$Z := X + C + \frac{\frac{E * F}{L + T}}{2^{X + Y} + 3^{X^2}} +$$

2. For the following formulae list the order of operations in their evaluation as in (2.1 thru 2.7):

Y := Y + Z * R * PHI * (A + B)

Z := X23/Y27 * Z48/L + 1

W := P↑R↑3 + W/L/M

3 The following charts each evaluate a single formula. Find them.

$$\begin{tikzcd}[row sep=small]
{} \arrow[d] & & {} \arrow[d] \\
\boxed{p \leftarrow A + R} \arrow[d] & & \boxed{p \leftarrow Z + 1} \arrow[d] \\
\boxed{\ell \leftarrow B - W} \arrow[d] & & \boxed{X \leftarrow p * X} \arrow[d] \\
\boxed{C \leftarrow \ell + T} \arrow[d] & & \boxed{p \leftarrow X + p} \arrow[d] \\
\boxed{p \leftarrow p * C} \arrow[d] & & \boxed{Z \leftarrow p / X} \arrow[d] \\
\boxed{C \leftarrow C / \ell} \arrow[d] & & \boxed{Z \leftarrow Z - p} \arrow[d] \\
\boxed{M \leftarrow A + T} \arrow[d] & & \boxed{p \leftarrow X / Z} \arrow[d] \\
\boxed{M \leftarrow M + p} \arrow[d] & & \boxed{X \leftarrow p + X} \\
\boxed{X \leftarrow C \uparrow M} & &
\end{tikzcd}$$

In exercise 3, the flow charts can be easily converted to $20^{\wedge}L$ code, with the possible exception of the computation with the operator $\uparrow$. We defer its analysis. Note that the charts can still not be converted into code until the nature of the operands and operators is clarified as to whether integer or fractional, since conversion may be required.

This clarification proceeds as follows:

The analysis should be based on a cascaded treatment of expressions. The simplest expression is that containing one operand E_2, e.g., X, 3.4, 26, MJ, etc. Then the natural definition is:

The arithmetic of an E_1 is that of its operand.

Now how is the arithmetic of an operand specified? It is natural that its arithmetic be unchanged during a computation, so the rule is:

The arithmetic of an operand named by an identifier is declared by the programmer, not by virtue of its identifier form. Such a declaration will have the form:

	real	X, MJ, TIPS, ROOO5	etc.
and	<u>integer</u>	ZZ, I4, P1	etc.

Operands named by numbers are defined to be those numbers. Any such number consisting of digits alone is understood to be an integer, e.g., 11, -7, 1210; but not 21.4, 0056, 1.5 $_{10}$8, 2$_{10}$7.

These latter are representations of fractional numbers.

Now if both simple operands, θ_1 and θ_2, are of the same arithmetic, that of the expression formed from them Θ_1 op Θ_2 would reasonably have that arithmetic. However the division of two integers does not necessarily produce an integer. Nor does exponentiation, as for example, 3↑ -7. An acceptable solution is to have two division operators possessing the sign / and ÷ . The former always yields a fractional number while the latter always yields an integer. But what integer, e.g., 3 ÷ 7, 25 ÷ 2, - 47 ÷ 8? Convention has established the definition:

$$a \div b = \text{sign}\ (a/b) \left[\ |a/b|\ \right].$$

Thus for the above, 0, 12, -5. While use of / would give (in **a typical machine** representation) .4285714350, 1250000052, - 5875000051.

Exponentiation may be treated as follows for a ↑ b.

```
        If b is an integer and
            if  b>0, then a * a ... * a (b times) and consequently of
the same arithmetic type as a.
            if b = 0, and if a ≠ 0  then 1 of the same type as a
else undefined.
            if b<0, and if a ≠ 0, then 1/(a*a...*a), (The denominator
has -b factors) and is of type real       else undefined.
        If b is real        and
            if  a>0 then  exp (b* ln(a)) of type fractional
            if  a=0, then  if  b>0, 0 of type fractional, else
if b ≤ undefined
            if a < 0, always undefined.
```

The use of conditional expressions is very natural in some instances. Thus :

$$y := \underline{\text{if}}\ x \leq 0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ (x+1)$$

defines y as the discontinuous function

$$y = \begin{cases} 0 & x \leq 0 \\ x+1 & x > 0 \end{cases}$$

Similarly

$$b := (\underline{\text{if}}\ x \leq 0\ \underline{\text{then}}\ c \wedge x > y\ \underline{\text{else}}\ g \wedge q \vee \neg (x \geq 7)) \vee g$$

defines (Boolean) b in terms of a conditional expression.

It is worthwhile to carry out by hand the following computations using expressions

```
real  ri, ra, rb ;
integer  n, i, j  ;
n  := 5 ;
ri  := n/(n + 15) ;
rb  := n + 6/(6* rl + 0.5) ;
i   := n := n - 2 ;
j   := rb - i ;
ra  := (j - 1) * rl * (rb - 4) ;
rl  := ra + rb + n + i + j + 8 * rl ;
rb  := (rl - rb * n + j - ra) ↑ (rb - j) + ra ;
j   := n := 1 + n + (j - 2) ;
i   := n + ra ;
```

The following example utilises Boolean expressions

```
real  ra, rb ;
integer  ia ;
Boolean  ba, bb ;
ra  := 7.5 ;
ia  := 5 ;
rb  := 3 * ra - 2 * ia ;
ba  := rb > ia   ia > ra ;
ra  := 2 * (ra - ia) - 1 ;
ba  :=    ra > ia   ba ;
bb  := (ba ≡ rb > ia) ra < rb ;
ba  :=    (ba   bb) ;
```

The following statements generate a sequence of values for SUM. Find the first four of these values.

```
       real  p, q, SUM ;
       integer  n ;
       n  := 1 ;
       p  := 0.5 ;
       SUM := 0 ;
       q  := 1 ;
loop:  SUM := SUM + q/n ;
       q  := q * p ;
       n  := n + 1 ;
       go to loop ;
```

5. Program Units

In a program there will be statements and declarations, Section 4.1.1 of Part I gives the important rules of how to join statements and declarations together to form a program. The main difficulty of this section is that of punctuation, particularly of when to write semi-colon and when not to. The difficulty is directly connected with the use of the delimiter end. As a guide the relevant rules may be restated as follows:

PUNCTUATION RULE 1: The first symbol following any statement (whether basic or not) must be one of the following three:

;   else   end

PUNCTUATION RULE 2: Any sequence ... end end end ... must always be terminated by semi-colon or else.

Punctuation rule 1 follows directly from the syntactic rules governing statements (Sections 4.1.1, 4.5.1, 4.6.1, and 5.4.1). Punctuation rule 2 follows from observing that an end, whenever it occurs, is the last symbol of some statement, and then applying punctuation rule 1.

5.1 The concept of block structure.

Block structure is critical to 20^L for it allows the efficient use of storage through overlay. Critical to its understanding is the concept of global and local relative to a block.

The concept local may be illustrated by an example of a program structure as follows:

```
 1:  begin real  A, B, C ;
         • • • • •
 2:  P:  A := B + 2 * C ;
         • • • • •
 3:      begin real A, D ;
           • • • • •
 4:      Q:  A := 2 * B + C ;
 5:          D := 2 + B + A ;
             • • • • •
 6:      P:  C := 2 * A - D ;
             • • • • •
 7:          go to P ;
             • • • • •
 8:          go to R ;
             • • • • •
 9:      end ;
         • • • • •
10:  R:  go to P ;
         • • • • •
11:  end
```

Here we have a larger block, from 1 to 11, containing as one statement a smaller block from 3 to 9. In the outer block we work with the identifiers A, B, and C, which are local to this block. In the statement at 2 a value is assigned to this A. The inner block introduces a new, local, A and a D. This A, then, has no relation to the A of the outer block, which is now screened. The variables B and C, on the other hand, are the same in both blocks. At 4 they are used to assign a value to the local A. This value is again used to assign a value to the local D at 5. These operations make no use whatsoever of the A of the outer block. At 6 a value is assigned to the non-local C, using the local A and D. Labels are automatically local. Thus the labels Q and P at 4 and 6 are only accessible from inside the inner block. The <u>go to</u> statement at 7 will therefore lead to the statement at 6. The <u>go to</u> statement at 8, on the other hand, will lead out of the inner block to 10 because the identifier R, being not declared in the inner block, will be non-local. The moment this passage out of the inner block occurs the local variables A and D are completely lost. The go to statement at 10 will lead to 2 because the label P at 6 is local to the inner block and thus inaccessible from 10.

Using the above example follow the action of the following program and find the values of those variables which are defined at the label STOP.

```
          begin  real  W, S, B, C ;
     1:   W  := 8 ;
     2:   S  := 3 ;
     3:   B  := 2 * W - S ;
     4:   C  := B - W ;
          begin real  P, W ;
     5:           W  := B - 2 * C ;
     6:           P  := C   2 - B ;
     7:   AA:     W  := P - 2 * W ;
     8:           C  := C + 1 ;
     9:           if W > 1 then go to AA ;
    10:           S  := W - P + S
          end
    11:   W  := W - C + S ;
          STOP:
          end
```

The scope of a label comprises, so to speak, all those statements from which the label may be seen.

The concept of scope may be illustrated by the example given. The scopes of the different quantities are as follows:

	Scope includes statements at
A and P in outer block	2, 10
B C, and R	2, 4, 5, 6, 7, 8, 10
D, Q, and A and P in inner block	4, 5, 6, 7, 8

An important step in the planning of 20^L program is the subdivision of the process into parts which may conveniently be written as blocks or procedures. In order to be able to do this the programmer must have a clear idea of the properties of these units.

Blocks are useful for expressing such parts of the program which form a closed process. A block is indispensable if in a process an array is needed whose size depends on the results of previous calculations. Such an array must then be local to a block. In addition any other quantity (simple variable, label, switch, procedure) which is used only internally during the work of the block, but which is not useful when the block is completed may be declared to be local to the block. This is particularly useful when different blocks of a program are written by different programmers. By using blocks the programmers will only have to agree on the non-local identifiers of the blocks, while inside each block the programmer is free to choose the identifiers of working quantities.

Procedures have two other uses:

Abbreviation of small ad-hoc functions; and a form of communication of closed processes between programs.

In particular they offer the optim of recursive definitions of processes.

Any block may be converted into a procedure by adding a heading to it. The heading will attach an identifier to the block and usually make some or all of the non-local identifiers formal parameters. Where the block in question is written specially for the program this conversion may be efficient only if the mechanism of the block is used two or more times with different non-local quantities, corresponding to two or more calls of the procedure, since a call of a procedure is a more elaborate process than a simple entrance into the corresponding block.

Frequently the formulae of a program may be shortened through the use of suitable function designators. As above this will be economical only if the corresponding ad-hoc procedure is used more than once during the program.

An example of the use of chained procedure calls is furnished by the following:

$$\text{Compute } y = \int_a^b H(x)\,dx$$

$$\text{where} \quad H(x) = \int_{T_1(x)}^{T_2(x)} G(x,t)\,dt$$

$$\text{and} \quad G(x,t) \equiv \sqrt{\sin^2(xt) + \cos^2(\,(1-x)\,t)}$$

If the procedure SIMPS. (F, L, U, Δ)

Evaluates the integral of the function named F from L to U to a precision Δ then the program would be as follows:

```
begin real  a, b, y ;
      real procedure  H(p); real p ;
begin real procedure  R := p   2 + 3 x p ;
      real procedure  R := p   3 - 2 x p ;
      real procedure  G(n) ; real n ;
      begin  G := sqrt ((sin (p x n))  2 +(cos((1 - p) x n ))   2 )  end G ;
             H := SIMPS ( G, R, S, .5_10-4)  end H ;
      real procedure  SIMPS ( F, L, U,    )
                 { 20^ L code for a Simpson's rule
                 {        program
             y := SIMPS ( H, a, b, .5_10-4)  end y.
```

6. Declarations

Declarations are really quite straight forward except in the case of arrays, procedures, switches, and macros.

6.1. Arrays

The detailed explanations of Sections 4.2.3.1 - 4.2.3.3 are relevant in a case like:

```
        real n ; array A[1 : 10] ;
        n := 2 ;
        A[n + 1] := n := n + 2 ;
```

Section 4.2.3.1 of Part I produces:

```
        A[3] := n :=
```

Section 4.2.3.2 of Part I gives the value of the expression as 4.

Section 4.2.3.3 of Part I assigns 4 to n and A[3].

Following the code given beow will clarify the consequences of non-dynamic arrays

```
    begin integer  i, j ; integer array  A[1:3, 1:2], C[0:2] ;
    j := i := 1 ;
    C[J-1] := A[j,i] := j := i + 2 * j + 2 ;
    A[2*i, C[j-2-3*i] - 3] := j - 2 * i ;
    C[A [2,2*j-8] -3] := i := k + j  ;
    A[C[j-i+1]/2, 4*A[1,1] -3*i] := A[1, 2*(i-j)] := A[2,2] - A[1,1] ;
    i :=  - A[3,2] ;
    j := i - j ;
    A[i, -j-2] ;6 C[i-1] := 7 ;
    A[A[2,2], C[1] - C[0]] := C[i] := 2 * i ;
    STOP:   end
```

Dynamic arrays are useful in matrix routines. Thus a declaration real array X[1: n, 1: n] allows the block in which the declaration is imbedded to use only storage necessary, say, for a linear equation routine.

6.2. Procedures

In the case of procedure declarations the following should be noted:

(i) The procedure declared is part of the block in which it is declared so its non-declared variables may 'communicate' with those in the block in which it is imbedded.

(ii) In the case of parameters not called by value - they are evaluated anew every time they are called. Whereas parameters called by value are evaluated once at the very beginning of each execution of the procedure.

(iii) Exits from a procedure involve a return to blocks and consequently may cause re-initialising of storage assignments. Thus in:

```
begin real x, j
      array A[1: 5]
      j := j + 1
begin real B
     array A[1: j]
     procedure M(p) real p
        begin integer i
                 • • •
           if x < 0 then A[i] := x else go to L
                 • • •
                          end end ;
  L :  A[i] := x  end ;
```

two different array elements are assigned the value x.

Switches are similarly tricky.

Thus, the kind of situation referred to by the remark of Section 5.3.5 of Part I may be illustrated by the following example:

```
begin switch  W := tt, Q[n + 2] ;
      switch Q := Q1, Q2, Q3 ;
      . . . . .
      A:  begin real n ;
                . . . . .
          TT:   go to W[2] ;
                . . . . .
                end block A
end
```

The go to statement at TT refers to W[2]. The designational expression for W[2] is Q[n+2]. Into this expression the variable n enters. Owing to the declaration real n in the head of block A the statement TT is outside the scope of the n of Q[n+2]. Consequently the go to statement is undefined.

As an exercise follow the action of the following statements and write a list of the labels to which the go to statements successively refer and find the final values of the variables:

```
begin integer n, s ;
   switch S := SB, S2, S3, STOP ;
   switch W := TW, S[n - s + y] ;
          n := 7 ;
   TW:    go to S[n - 4] ;
   SB:    n := n - 1 ;
          s := s + n ;
          go to W[n - 2] ;
   S3:    n := n - 2 ;
          s := n - 2 ;
          go to W[n - s - 1]
   STOP:
   end
```

Macros are particularly simple to understand. They are, after all, blocks of code which are substituted into the code wherever they are called. After substitution their effect is precisely as though they had been put there originally by the programmer, one must be careful about the rules for substitution of parameters. They are:

(i) A formal macro parameter can only be an'identifier'. For all occurrences of that identifier in a macro definition, an instance of a macro call will cause that identifier to be replaced in a copy of the definition by the actual parameter. Thus the replacement of the formal parameter x by the string x x will not create a non-terminating string: x x ... x ... since the replacement is into a copy of the definition.

(ii) The actual parameter, if delimited by a set of matching parentheses replaces the corresponding formal parameter with the outermost parentheses stripped.

Thus:

```
macro   S( A, B, C)
  x: = A + t
  B
  T| Q + ( C )
  end
```

A call occurring in the code:

```
S (sin (p * x), (S (sin (p * x),  y := cos(p * x), (R + 2) ), ((( 3 )))
```

will cause

```
firstly   x := sin (p * x) + t
          S (sin (p * x), y := cos(P * x), (R + 2)
          T|Q + ((( 3 )))
```

then:

then:

```
          x := sin (p * x)  + t
```

finally:

```
          x := sin (p * x) + t
          y := cos (p * x)
          T | Q + (R + 2)
          T | Q + ((( 3 )))
```

will be produced. Notice that parentheses serve three delimiting functions in this example.

7. For Statements

For statements are the most complex control statement in the language 20 L. Finding the values assigned to the controlled variable in the following for statements and the final value of s:

```
       begin real p, q, r, s ; integer k, m ;
       p := 1 ;   q := 2 ;   r := 3 ;   s := 0 ;
       for k := p + q, q - p, r * p  - q do s := s + k ;
       for m := q step r until 7 * q do s := s - m ;
       for k := 2, s, 2 step 2 until 6 do s := s + 2 * k ;
       for m := s + 45, m + 2 while s < 0 do s := s - m ;
       for k := 1 step 1 until 5 do
       for m := 3 step - 1 until 0 do s := s + k + m ;
```

will clarify the concept of the for statement considerably.

For statements are particularly useful for executing operations on vectors and matrices (described in 20^L as arrays). A simple example is the addition of two vectors VA and VB to give a third VC. This may be expressed as

```
     array VA, VB, BC [1: n]; integer i ;
     for i := 1 step 1 until n do VC[i] := VA[i] 2 VB[i] ;
```

Note that the quantity n cannot be declared in the same block head as the arrays VA, VB, VC (cf. Section 5.2.4.2 of Part I).

8. Lists

T lists have been reported on elsewhere. Examples of their use are given in the article in "Communications of ACM" (April 1960).

Lists are represented by X [a, b] where:

X is the name of the list; a is p, / , or \/ meaning prefix, left, or right-site section, respectively ; b is ¢ (current) or * .

p has a 3-bit form g f e :

$$e = \begin{cases} 0 \text{ element} \\ 1 \text{ list} \end{cases}$$

$$f = \begin{cases} 0 \text{ direct} \\ 1 \text{ indirect} \end{cases}$$

$$g = \begin{cases} 0 \text{ non-terminal} \\ 1 \text{ terminal} \end{cases}$$

Thus p = 101(2) or p = 5(10) means a direct terminal-list entry. The indirect means referral to a word having structure not sequenced by Tlist sequencing modes. In particular, the referencing may be to arrays declared within 20 . The empty list is represented by (,) and the list:

(, {2.X} , (, {2.Z[1,2]} , {3.Y}) , {1.a})

refers to 20 L objects X, Y, and Z[1,2] (arrays, variables, lists, procedures, etc.; while a itself is stored in the list. The arrow is omitted in the above representation.

As an example consider the procedure

```
procedure equal ( x, y, E); list x, y; Boolean E
begin  seqw (x, exit); seqw (y, exit);  E := true ;
      1 :   if  x[p,*] = y[p,*]
            then begin
            if  x [/, ¢] = y[ /, ¢]
               then  go to 1
            else if  x[p,¢] = 1
```

```
                    then go to 1 end
                  E := false
              exit: ; end equal.
```

which checks if two lists are equal.

Bibliography and Note

1. "Report on the Algorithmic Language ALGOL 60", Communications of the ACM, 5 299-314, May (1960)

2. "TASS, an An Assembly System for the IBM 650 Drum Calculator with Disk Storage", Computation Center, Carnegie Institute of Technology, 1959.

3. "Symbol Manipulation by Threaded Lists", A.J.Perlis and C.Thornton, Communications of the ACM, 4, 184-194, April (1960).

4. 20 $\hat{}$ stands for G-20 Carnegie Tech Algebraic Translator.

5. "Bendix G-20 System" Communications of the ACM, 5, 325-329, May 1960

6. Several of the examples in this section are due to staff members of the Danish Regnecentralen.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

For Algebraic Language See p. I-2 through I-8
For Symbolic Coding and List Processing See p. I-9

All references are given through section numbers. The references are given in three groups:

def: Following the abbreviation "def", reference to the syntactic definition (if any) is given.

synt: Following the abbreviation "synt", references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text: Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined words have been collected at the beginning.

The examples have been ignored in compiling the index.

$+$, see: plus

$-$, see: minus

$\times$, see multiply

$/$, $\div$, see divide

$\uparrow$, see: exponentiation

$<, \leq, =, \geq, >, \neq$ see: (relational operator)

$\equiv, \supset, \vee, \wedge, \neg$ see: (logical operator)

, see: comma

., see: decimal point

$_{10}$, see: ten

: - see: colon

;, see: semicolon

:= see: colon equal

␣, see: space

( ), see: parentheses

[ ], see: subscript bracket

‘ ’, see: string quote

(actual parameter), def 3.2.1,4.7.1

(actual parameter list), def 3.2.1, 4.7.1

(actual parameter part), def 3.2.1, 4.7.1

(adding operator), def 3.3.1

alphabet, text 2.1

arithmetic text 3.3.6

(arithmetic expression), def 3.3.1 synt 3,3.1.1, 3.3.1, 3.4.1, 4.2.1,
4.6.1, 5.2.1 text 3.3.3

(arithmetic operator), def 2,3 text 3.3.4

array, text 3.1.4.1

array, synt 2.3, 5.2.1, 5.4.1

(array declaration), def 5.2.1 synt 5 text 5.2.3

(array identifier), def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8

(array list) def. 5.2.1

(array segment) def 5.2.1

(assignment statement), def 4.2.1 synt 4.1.1 text 1, 4.2.3

(basic statement), def 4.1.1 synt 4.5.1

(basic symbol) def. 2

begin, synt 2.3, 4.1.1

(block), def 4.1.1 synt 4.5.1 text 1, 4.1.3;5

(block head), def 4.1.1

Boolean, synt 2.3, 5.1.1 text 5.1.3

(Boolean expression) def 3.4.1 synt 3, 3.3.1, 4.2.1, 4.5.1, 4.6.1,

text 3,4.3

(Boolean factor), def 3.4.1

(Boolean primary) def 3.4.1

(Boolean secondary), def 3.4.1

(Boolean term), def 3.4.1

(bound pair), def 5.2.1

(bound pair list) def 5.2.1

(bracket) def 2.3

(code), synt 5.4.1 text 4.7.8 5.4.6

colon :, synt 2.3, 3.2.1, 4.1.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1

colon equal :=, synt 2.3, 4.2.1, 4.6.1, 5.3.1

comma , , synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1

comment, synt 2.3

comment convention, text 2.3

(compound statement) def. 4.1.1 synt 4.5.1 text 1

(compound tail), def. 4.1.1

(conditional statement), def 4.5.1 synt 4.1.1 text 4.5.3

(decimal fraction) def 2.5.1

(decimal number), def 2.5.1 text 2.5.3

decimal point ., synt 2.3, 2.5.1

(declaration), def 5 synt 4.1.1 text 1, 5 (complete section)

(declarator), def 2.3

(delimiter) def 2.3 synt 2

(designational expression), def 3.6.1 synt 3, 4.3.1, 5.3.1 text 3.6.3

(digit), def 2.2.1 synt 2, 2.4.1, 2.5.1

dimension, text 5.2.3.2

divide / ÷ , synt 2.3, 3.3.1 text 3.3.4.2

do, synt 2.3, 4.6.1

(dummy statement) def 4.4.1 synt 4.1.1 text 4.4.3

else, synt 2.3, 3.3.1, 3.4.1, 3.5.1, 4.5.1 text 4.5.3.2

(empty), def 1.1 synt 2.6.1, 3.2.1, 4.4.1, 4.7.1, 5.4.1

end, synt 2.3, 4.1.1

entier, text 3.2.5

exponentiation , synt 2.3, 3.3.1 text 3.3.43

(exponent part), def 2.5.1 text 2.5.3

(expression), def 3 synt 3.2.1, 4.7.1 text 3 (corplete section)

(factor), def 3.3.1

false, synt 2.2.2

for, synt 2.3, 4.6.1

(for clause), def 4.6.1 text 4.6.3

(for list), def 4.6.1 text 4.6. 4

(for list element), def 4.6.1 text 4.6.4.1, 4.6.4.2, 4.6.4.3

(formal parameter), def 5.4.1 text 5.4.3

(formal parameter list), def 5.4.1
(formal parameter part,)def 5.4.1
(for statement) def 4.6.1 synt 4.1.1, 4.5.1 text 4.6 complete section)

(function designator), def 3.2.1 synt 3.3.1, 3.4.1 text 3.2.3, 5.4.4.

go to, synt 2.3, 4.3.1

(go to statement) def 4.3.1 synt 4.1.1 text 43.3

(identifier) def 2.4.1 synt 3.1.1, 3.2.1, 3.5.1, 5.4.1 text 2.4.3

(identifier list), def 5.4.1

if, synt 2.3, 3.3.1, 4.5.1

(if clause) def 3.3.1, 4.5.1 synt 3.4.1, 3.5.1 text 3.3.3, 4.5.3.2

(if statement), def 4.5.1 text 4.5.3.1

(implication), def 3.4.1

integer, synt 2.3, 5.1.1 text 5.1.3

(integer), def 2.5.1 text 2.5.4

label, synt 2.3, 5.4.1

(label) def 3.5.1 synt 4.1.1, 4.5.1, 4.6.1 text 1 4.1.3

(left part), def. 4.2.1

(left part list), def 4.2.1

(letter) def 2.1 synt 2, 2.4.1, 3.2.1, 4.7.1

(letter string), def 3.2.1, 4.7.1

local, text 4.1.3

(local or own type), def 5.1.1 synt 5.2.1

(logical operator), def 2.3 synt 3.4.1 text 3.4.5

(logical value), def. 2.2.2 synt 2, 3.4.1

(lower bound) def 5.2.1 text 5.2.4

minus − , synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1

multiply X, synt 2.3, 3.3.1 text 3.3.4.1

(multiplying operator), def 3.3.1

Nonlocal, text 4.1.3

(number), def 2.5.1 text 2.5.3, 2.5.4

(open string), def 2.6.1

(operator) def 2.3

own, synt 2.3, 5.1.1 text 5, 5.2.5

(parameter delimiter) def 3.2.1, 4.7.1 synt 5.4.1 text 4.7.7

parentheses ( ) synt 2.3, 3.2.1, 3.3.1, 3.4.1, 3.5.1, 4.7.1, 5.4.1 text 3.3.5.2

plus +, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1

(primary), def 3.3.1

procedure, synt 2.3, [illegible].4.1

(procedure body), def 5.4.1

(procedure declaration) def 5.4.1 synt 5 test 5.4.3

(procedure heading), def 5.4.1 text 5.4.3

(procedure identifier) def 3.2.1 synt 3.2.1, 4.7.1, 5.4.1 text 4.7.5.4

(procedure statement), def 4.7.1 synt 4.1.1 text 4.7.3

program, text 1

(proper string), def 2.6.1

quantity, text 2.7

real, synt 2.3, 5.1.1 text 5.1.3

(relation) def 3.4.1 text 3.4.5

(relational operator) def 2.3., 3.4.1

scope, text 2.7

semicolon; synt 2.3, 4.1.1, 5.4.1

(separator) def 2.3

(sequential operator) def 2.3

(simple arithmetic expression) def 3.3.1 text 3.3.3

(simple Boolean) def 3.4.1

(simple designational expression) def 3.6.1

(simple variable), def 3.1.1 synt 5.1.1 text 2.4.3

space #, synt 2.3 text 2.3, 2.6. 3

(specification part) def 5.4.1 text 5.4.5

(specificator) def 2.3

(specifier) def 5.4.1

standard function, text 3.2.4, 3.2.5

(statement) def 4.1.1, synt 4.5.1, 4.6.1, 5.4.1 text 4 (complete section)

statement bracket, see: begin end

step, synt 2.3, 4.6.1 text 4.6.4.2

string, synt 2.3, 5.4.1

(string) def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3

subscript, text 3.1.4.1

subscript bound, text 5.2.3.1

subscript bracket synt 2.3, 3.1.1, 3.5.1, 5.2.1

(subscript variable), def 3.1.1 text 3.1.4.1

(subscript expression) def 3.1.1 synt 3.6.1

(subscript list), def 3.1.1

successor, text 4

switch , synt 2.3, 5.3.1, 5.4.1

(switch declaration) def 5.3.1 synt 5 text 5.3.3

(switch desigrator) def 3.6.1 text 3.6.3

(switch identifier) def 3.6.1 synt 3.2.1, 4.7.1, 5.3.1

(switch list) def 5.3.1

(term), def 3.3.1

ten synt 2.3. 2.5.1

then, synt 2.3, 3.3.1, 4.5.1

transfer function, text 3.2.5

true, synt 2.2.2

(type) def 5.1.1 synt 5.4.1 text 2.8

(type declaration), def 5.1.1 synt 5 text 5.1.3

(type list) def 5.1:1

(unconditional statement) def 4.1.1, 4.5.1

(unlabelled basic statement), def 4.1.1

(unlabelled block) def 4.1.1

(unlabelled compound) def 4.1.1

(unsigned integer) def 2.5.1, 3.5.1

(unsigned number), def 2.5.1 synt 3.3.1

until, synt 2.3, 4.6.1 text 4.6.4.2

(upper bound), def 5.2.1 text 5.2.4

value, synt 2.3, 5.4.1

value, text 2.8, 3.3.3

(value part), def 5.4.1 text 4.7.3.1

(variable) def 3.1.1 synt 3.3.1, 3.4.1, 4.2.1, 4.6.1 text 3.13

(variable identifier), def 3.1.1

while, synt 2.3, 4.6.1 text 4.6.4.3

The above portion of the Index is adapted from "Report on the Algorithmic Language ALGOL 60" which appeared in Communications of the ACM, Vol 3, No. 5, May 1960.

SYMBOLIC CODING AND LIST PROCESSING

address chain , def 3.8.1

address expression , def 3.7

address interpreter , def 4.8.1

code line , def 4.8.1

copy, text 3.2.4.2

elementary address , def 3.7.1

equivalent declaration def 5.6.1

indirect address expression , synt 3.7.1

insert, text 3.2.4.2

instruction designator def 4.8.1

library declaration def 5.7.1

list, text 3.2.4.2

list procedures, text 3.2.4.2

list subscript expression, text 3.8

logical expression , def 3.5

macro declaration , def 5.5.1 text 5.5.3

macro designator , def 4.8.1

macro identifier , synt 4.8.1

macro statement , def 4.9.9

next , text 3.2.4.2

operation designator def 4.8.1

pair list def 5.6.1

seq. , text 3.2.4.2

shift measure , def 3.5.1

AD________________DIV____________

Carnegie Institute of Technology, Pittsburgh, Pennsylvania.
RESEARCH AND DEVELOPMENT IN PROGRAMMING STUDY ASSIGNMENTS by A. J. Perlis. Part I-Final Report; Part II-Flow Charts; Part III-Programming Manual. Feb 1961, 132 pages. Contract No. DA-36-039 SC-75081 Unclassified Report also ERRATA AND ADDENDA TO THE FINAL REPORT, in a separate volume; 13 p.; Oct 1961 also SOME NOTES ON THE PROCESSOR 20 P, in a separate volume; 11 p.; Oct 1961.

A single programming language is described, through it has from the programmer's point of view three forms. The forms are: (i) An algebraic language, via ALGOL; (ii) a symbolic machine-like coding language, like TASS; (iii) a symbolic list processing language, like Threaded Lists. The three are described as one language, 20 L, using the same syntax description, in Part I. There is only one processor, 20 P, described in Part II. The design of the processor is described in terms of actions taken at various phases of the translation process.

UNCLASSIFIED

1. Programming
2. Automatic Programming
3. Algebraic Compilers
4. ALGOL
I. Perlis, A. J.
II. U.S. Army R&D Laboratory Fort Monmouth, N. J.
III. Contract DA-36-039 SC-75081

AD________________DIV____________

Carnegie Institute of Technology, Pittsburgh, Pennsylvania.
RESEARCH AND DEVELOPMENT IN PROGRAMMING STUDY ASSIGNMENTS by A. J. Perlis. Part I-Final Report; Part II-Flow Charts; Part III-Programming Manual. Feb 1961, 132 pages. Contract No. DA-36-039 SC-75081 Unclassified Report also ERRATA AND ADDENDA TO THE FINAL REPORT, in a separate volume; 13 p.; Oct 1961 also SOME NOTES ON THE PROCESSOR 20 P, in a separate volume; 11 p.; Oct 1961.

A single programming language is described, through it has from the programmer's point of view three forms. The forms are: (i) An algebraic language, via ALGOL; (ii) a symbolic machine-like coding language, like TASS; (iii) a symbolic list processing language, like Threaded Lists. The three are described as one language, 20 L, using the same syntax description, in Part I. There is only one processor, 20 P, described in Part II. The design of the processor is described in terms of actions taken at various phases of the translation process.

UNCLASSIFIED

1. Programming
2. Automatic Programming
3. Algebraic Compilers
4. ALGOL
I. Perlis, A. J.
II. U.S. Army R&D Laboratory Fort Monmouth, N. J.
III. Contract DA-36-039 SC-75081

AD________________DIV____________

Carnegie Institute of Technology, Pittsburgh, Pennsylvania.
RESEARCH AND DEVELOPMENT IN PROGRAMMING STUDY ASSIGNMENTS by A. J. Perlis. Part I-Final Report; Part II-Flow Charts; Part III-Programming Manual. Feb 1961, 132 pages. Contract No. DA-36-039 SC-75081 Unclassified Report also ERRATA AND ADDENDA TO THE FINAL REPORT, in a separate volume; 13 p.; Oct 1961 also SOME NOTES ON THE PROCESSOR 20 P, in a separate volume; 11 p.; Oct 1961.

A single programming language is described, through it has from the programmer's point of view three forms. The forms are: (i) An algebraic language, via ALGOL; (ii) a symbolic machine-like coding language, like TASS; (iii) a symbolic list processing language, like Threaded Lists. The three are described as one language, 20 L, using the same syntax description, in Part I. There is only one processor, 20 P, described in Part II. The design of the processor is described in terms of actions taken at various phases of the translation process.

UNCLASSIFIED

1. Programming
2. Automatic Programming
3. Algebraic Compilers
4. ALGOL
I. Perlis, A. J.
II. U.S. Army R&D Laboratory Fort Monmouth, N. J.
III. Contract DA-36-039 SC-75081

AD________________DIV____________

Carnegie Institute of Technology, Pittsburgh, Pennsylvania.
RESEARCH AND DEVELOPMENT IN PROGRAMMING STUDY ASSIGNMENTS by A. J. Perlis. Part I-Final Report; Part II-Flow Charts; Part III-Programming Manual. Feb 1961, 132 pages. Contract No. DA-36-039 SC-75081 Unclassified Report also ERRATA AND ADDENDA TO THE FINAL REPORT, in a separate volume; 13 p.; Oct 1961 also SOME NOTES ON THE PROCESSOR 20 P, in a separate volume; 11 p.; Oct 1961.

A single programming language is described, through it has from the programmer's point of view three forms. The forms are: (i) An algebraic language, via ALGOL; (ii) a symbolic machine-like coding language, like TASS; (iii) a symbolic list processing language, like Threaded Lists. The three are described as one language, 20 L, using the same syntax description, in Part I. There is only one processor, 20 P, described in Part II. The design of the processor is described in terms of actions taken at various phases of the translation process.

UNCLASSIFIED

1. Programming
2. Automatic Programming
3. Algebraic Compilers
4. ALGOL
I. Perlis, A. J.
II. U.S. Army R&D Laboratory Fort Monmouth, N. J.
III. Contract DA-36-039 SC-75081